File No. S360-29
Form C28-6571-0

IBM Systems Reference Library

IBM Operating System/360

PL/I: Language Specifications

This manual is a description of the full facilities of PL/I to be implemented under Operating System/360. However, the reader should not assume that all facilities will be available at initial release. Manuals for specific System/360 implementations will be released later.

Another publication will be issued specifying a subset of the facilities of the language described in this manual. This subset is planned for implementation under the Basic Operating System/360 and Basic Program Support for System/360.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to the IBM Corporation, Programming Systems Publications, Department D39, 1271 Avenue of the Americas, New York, N. Y., 10020.

GOALS OF THE LANGUAGE

Throughout the relatively brief history of electronic data processing, certain computers have been identified with a particular field of activity, either commercial or scientific.

Programmers generally have specialized in one field or the other. High-level languages, of course, have emphasized this divergence, going in one direction for commercial programming and in another direction for scientific programming.

Until recently, this difference presented few problems. Each language was adequate for its use; the commercial programmer dealt with relatively few computations performed upon great amounts of data; the scientific programmer performed complex calculations using small amounts of data.

Now, however, the situation is changing. Business and industry have discovered new uses for the computer, and the commercial programmer finds himself concerned with more involved computations in statistical forecasting and in linear programming for operations research.

In science and engineering, the programmer needs a language to simplify the preparation of reports, to sort and edit technical data; he finds more need for input and output operations. The engineer specifically wants the ability to handle data at the bit level for applications such as circuit analysis.

Today's new computing systems have been designed to cope with all of these computing problems. They handle commercial and scientific programs with equal ease, with new power and new speed; they provide facilities for such new techniques as shared data processing, asynchronous program execution, and real-time processing.

None of the traditional high-level languages, however, can be used with efficiency across the entire range of ability of these new computers.

That is the reason for PL/I, a multipurpose programming language for use not only by commercial and scientific programmers but by the real-time programmer and the systems programmer as well. It is a language designed for efficiency, a language that enables the programmer to use virtually all the power of his computer.

PL/I is organized so that any programmer, no matter how extensive his experience, can use it easily at his own level.

This manual, because it is a reference manual of the entire language, shows the range and power of PL/I, its ability to handle the most complex computing problems.

Actually, however, PL/I need be no more complex than the program for which it is used.

One of the primary aims in the design of the language was modularity, that is, providing different subsets of the language for different applications and different levels of complexity. A programmer using one subset need not even know about the unused facilities.

Although PL/I is relatively machine independent, this modularity might be compared to a completely equipped data processing center. A novice programmer would use only a small part of the system; he can ignore the rest of the equipment. More complex programs, of course, would require more equipment. Some programs would use certain modules of equipment; other programs, other modules. Rarely, if ever, would a program require use of all the facilities.

In PL/I, every attribute -- or description -- of a variable, every option, and every specification has been given a "default" interpretation. Wherever the language provides for one or more alternatives, a "default" interpretation -- or assumption -- is made by the compiler if no choice is stated by the programmer. And in each case, the assumption that was chosen in the design of the language is the one most likely to be required by the programmer who need not know that alternatives exist.

The "modularity" and the "default" aspects are the bases upon which the simplicity of PL/I has been built. They are also part of its power.

rence of a digit, which may be 0 through 9 inclusive.
   b. file-name. This denotes the occurrence of the notation variable named file-name. An explanation of file-name is given elsewhere in the manual.
   c. DO-statement. This denotes the occurrence of a DO statement. The upper-case letters are used for emphasis.
2. A notation constant denotes the literal occurrence of the characters represented. A notation constant consists either of all capital letters or of a special character.

Example:

    DECLARE identifier FIXED;

This denotes the literal occurrence of the word DECLARE followed by the variable "identifier," which is defined elsewhere, followed by the literal occurrence of the word FIXED followed by the literal occurrence of the semicolon (;).
3. The term "syntactical unit," which is used in subsequent rules, is defined as one of the following:
   a. a single variable or constant, or
   b. any collection of variables, constants, syntax-language symbols, and reserved words surrounded by braces or brackets.
4. Braces { } are used to denote grouping.

Example:

$$\text{identifier} \left\{\begin{array}{l}\text{FIXED}\\ \text{FLOAT}\end{array}\right\}$$

The vertical stacking of syntactical units indicates that a choice is to be made. The above example indicates that the variable "identifier" must be followed by the literal occurrence of either the word FIXED or the word FLOAT.
5. The vertical stroke | indicates that a choice is to be made.

Example:

    identifier {FIXED|FLOAT}

This has exactly the same meaning as the above example. Both methods are used in this manual to display alternatives.
6. Square brackets [ ] denote options. Anything enclosed in brackets may appear one time or may not appear at all.

Example:

    CHARACTER (length) [VARYING]

This denotes the literal occurrence of the word CHARACTER followed by the variable "length" enclosed in parentheses and optionally followed by the literal occurrence of the word VARYING. If, in rule 4, the two alternatives also were optional, the vertical stacking would be within brackets, and there would be no need for braces.
7. Three dots ... denote the occurrence of the immediately preceding syntactical unit one or more times in succession.

Example:

    [digit] ...

The variable, "digit," may or may not occur since it is surrounded by brackets. If it does occur, it may be repeated one or more times.
8. Underlining is used to denote an element in the language being described when there is conflict between this element and one in the syntax language.

Example:

    operand {&|$\underline{|}$|¬} operand

This denotes that the variables "operand" are separated by either an "and" (&), an "or" (|), or a "not" (¬) symbol. The constant | is underlined in order to distinguish the "or" symbol in the PL/I language from the "or" symbols in the syntax language.
9. min max. The combination of these two words with associated numeric values specifies the minimum and maximum number of times a syntactical unit may occur. When min is used without max, the implied max is infinity. When max is used without min, the implied min is zero.

Examples:

   a. min 2 max 6 {digit|letter}

   This denotes that either "digit" or "letter" intermixed in any com-

bination must occur at least two times, but no more than six.

b. min 5 {digit|letter}

The variables "digit" or "letter" intermixed in any combination must occur at least five times, but there is no limit on the number of times over five that they may occur.

c. max 3 label

The variable "label" may not occur more than three times in succession. It may not be present at all, or it may occur one, two, or three times.

ELEMENTARY LANGUAGE STRUCTURE

PL/I allows the programmer to write the statements of his program in a free-field format. A statement, which is a string of characters, is always terminated by the special character, semicolon. A program which is, in turn, a sequence of statements, can thus be regarded simply as a single string of characters, with no special internal grouping. Hence, a PL/I program can be physically represented and transmitted to a computer in a natural way by means of almost any input medium, including the important case of a typewriter at a remote terminal.

Input conventions, depending upon the machine configuration or the compiler, can, of course, be set up so that the program string may be presented to the computer through the familiar medium of fixed-length records, e.g., punched cards, using certain predetermined fields of the records for the program string, and other fields for arbitrary purposes.

LANGUAGE CHARACTER SET

One of two character sets may be used to write a source program: either a 60-character set or a 48-character set. No assumptions are made in the language about external or internal codes for the characters. For a given program, the choice between the two sets is optional. (In practice, this choice will depend upon the equipment that is available.)

60-Character Set

The 60-character set is composed of English language alphabetic characters, digits, and special characters.

There are 29 alphabetic characters, letters A through Z and three additional characters that are defined as and treated as alphabetic characters. These characters and the graphics by which they are represented are:

Currency symbol	$
Commercial At-sign	@
Number sign	#

There are ten digits. Decimal digits are the digits 0 through 9. A binary digit (bit) is either a 0 or a 1.

An alphameric character is either an alphabetic character or a digit.

There are 21 special characters. The names and graphics by which they are represented are:

Name	Graphic
Blank	
Equal or Assignment symbol	=
Plus	+
Minus	-
Asterisk or Multiply symbol	*
Slash or Divide symbol	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point or Period	.
Quotation mark	'
Percent symbol	%
Semicolon	;
Colon	:
Not symbol	¬
And symbol	&
Or symbol	|
Greater Than symbol	>
Less Than symbol	<
Break character	_
Question mark	?

Special characters may be combined to create operators, e.g., >=, denoting "greater than or equal to"; ||, denoting concatenation.

48-Character Set

The characters making up the 48-character set are identical to those of the 60-character set, with restrictions and changes as described in Appendix 5.

DELIMITERS

Certain characters are used as delimiters and fall into three classes:

operators
parentheses
separators and other delimiters

Operators

Operators used by the language are divided into four types:

arithmetic operators
comparison operators
bit-string operators
string operators

Arithmetic Operators

The arithmetic operators are:

+	denoting addition or prefix plus
-	denoting subtraction or prefix minus
*	denoting multiplication
/	denoting division
**	denoting exponentiation

Comparison Operators

The comparison operators are:

>	denoting greater than
>=	denoting greater than or equal to
=	denoting equal to
¬=	denoting not equal to
<=	denoting less than or equal to
<	denoting less than

Bit-String Operators

The bit-string operators are:

¬	denoting not
&	denoting and
|	denoting or

String Operator

The string operator is:

||	denoting concatenation

Parentheses

Parentheses are used in expressions and for enclosing lists.

(	left parenthesis
)	right parenthesis

Separators and Other Delimiters

Name	Graphic	Use
comma	,	separates elements of list
semicolon	;	terminates statements
assignment symbol	=	used in assignment statement
colon	:	follows labels, used with dimension specifications
blank		used as a separator
quotation mark	'	encloses string constants
break character	_	used in identifiers
period	.	separates items in qualified names; used as a decimal or binary point in constants

DATA CHARACTER SET

Although the language character set is a fixed set defined for the language the data character set has not been limited. Data

may be represented by characters from the language set plus any other character permitted by the particular machine configuration.

Any character that will result in a unique pattern is a valid character in the data character set, and may be used in source programs to construct character-string constants, and comments.

COLLATING SEQUENCE

In the execution of PL/I programs, comparisons of character data will observe the collating sequence resulting from the representations of involved characters in bytes of System/360 storage, in extended binary coded decimal interchange code (EBCDIC).

The sequencing is, low to high, currency symbol, number sign, commercial at-sign, and the twenty-six English language alphabet letters A to Z. The number sign and the commercial at-sign are not used in the 48-character syntactic character set.

IDENTIFIERS

An <u>identifier</u> is a string of alphameric and break characters with the initial character always being alphabetic.

Identifiers in the language are used for:

- scalar variable names
- array names
- structure names
- statement labels
- entry names
- file names
- keywords
- task identifiers
- condition names
- headings for external names

Any number of break characters are allowed within an identifier; however, consecutive break characters are not permitted, nor can a break character be the final character of an identifier.

Examples:

```
A
BCD320
XR20A
RATE_OF_PAY
#32_45
$L32
X@_52
@531
AB12#
```

<u>Length of Identifiers</u>

Identifiers that a programmer constructs in writing a PL/I program must be composed of not more than 31 characters.

KEYWORDS

A <u>keyword</u> is an identifier used in the language and has a special meaning. Keywords are <u>not</u> reserved words. They may be classified:

- statement identifiers
- attributes
- separating keywords
- built-in function names
- options
- conditions

<u>Statement Identifiers</u>

A <u>statement identifier</u> is a keyword used to identify the nature of a statement (see "Simple Statements"). Some statement identifiers consist of more than one identifier, separated by blanks.

Examples:

```
GO TO
DECLARE
READ
```

Attributes

Attributes are keywords that specify characteristics to describe the nature of data, procedures, and other elements of the language.

Example:

```
FLOAT
RECURSIVE
SEQUENTIAL
```

Separating Keywords

The five separating keywords are used to separate parts of the IF and DO statements. They are THEN, ELSE, BY, TO, WHILE.

Built-in Function Names

A built-in function name is a keyword that is the name of an algorithm provided by the language and accessible to the programmer (see "Function References and Function Procedures" in this chapter).

Examples:

```
DATE
EXP
```

Options

An option is a specification that may be used by the programmer to influence the execution of a statement.

Examples:

```
TASK
CROSS
```

Conditions

A condition is a keyword used in the ON, SIGNAL, and REVERT statements, and as a prefix to other statements (see "Prefixes"). The programmer may specify special action on occurrence of the condition (see "Interrupt Operations").

Examples:

```
OVERFLOW
ZERODIVIDE
```

WORDS AND THE USE OF BLANKS

A word is an identifier, a constant, or a picture specification.

In those cases in the language where two words lie adjacent, and are not separated by an operator, an equal sign, a parenthesis, a colon, a comma, or a semicolon, the words must be separated by a blank or comment. One or more blanks may appear freely between words and adjacent delimiters. Blanks are not permitted within such composite operators as ** or >=.

Examples:

CALLA	is not equivalent to CALL A
9.6E+2	is not equivalent to 9.6E +2
A TO B BY C	is not equivalent to ATOBBYC
AB+BC	is equivalent to AB + BC

COMMENTS

General form:

```
/* character-string */
```

A comment may be used anywhere that a blank is permitted except in a character constant or a picture specification. The character string must not contain the character combination */ in that sequence.

Example:

```
LABEL: /* THE BLOCK OF CODING  BETWEEN
BEGIN-END IS USED FOR PAYROLL CALCULA-
TIONS */
     BEGIN;
     .
     .
     .
     END;
```

LOGICAL PROGRAM STRUCTURE

PROGRAM ELEMENTS

A PL/I program is constructed from program elements. The basic program element is the statement.

Statements are grouped into larger program elements, the group and the block. There are two types of statements: simple and compound.

Simple Statements

A simple statement is defined as:

```
[[statement-identifier]
statement-body] ;
```

The "statement identifier," if it appears, is a keyword , characterizing the kind of statement. If it does not appear, and the statement body does appear, the statement is then an assignment statement. If only the semicolon appears, the statement is called a null statement.

Examples:

```
DO I = J TO    (DO is the keyword)
  10;

A = B + C      (assignment statement)

  ;            (null statement)
```

Compound Statements

A compound statement is a statement that contains other program elements. There are only two of them. They are:

The IF compound statement

The ON compound statement

The final contained statement of a compound statement is a simple statement and thus has a terminal semicolon. Hence, the compound statement will automatically be terminated by this semicolon.

Examples:

```
IF A=B THEN GO TO S1; ELSE A=C;

ON OVERFLOW GO TO OVFIX;
```

Each PL/I statement is described in the alphabetic list of statements in Chapter 6.

Labels

Statements may be labeled to permit reference to them. A labeled statement has the form:

```
identifier:[identifier:]...statement
```

The one or more "identifiers" are called labels and may be used interchangeably to refer to that statement.

Labels appearing before PROCEDURE and ENTRY statements are special cases and are known as entry names (see "Procedure References"). All other labels are called statement labels.

A label appearing before a statement is said to be declared, by virtue of its appearance as a label.

Statement labels declared before DECLARE and IMPLICIT are ignored.

Groups

A group is a collection of one or more statements and is used for control purposes.

A group has one of two forms. The first form, called a DO group, is:

```
[label:] . . . DO-statement
               program-element-1
               program-element-2
                     .
                     .
                     .
               END [label];
```

The label following END must be one of the labels of the DO statement.

The DO statement is called the heading statement of the DO group, and may specify iteration.

The second form of a group is simply a single statement, as follows:

```
[label:] . . . statement
```

The "statement" is any kind of statement except DO, END, PROCEDURE, BEGIN, DECLARE, IMPLICIT, FORMAT, ENTRY, or any compile-time statement.

Example of the first form:

```
ALPHA: DO;
       A=B*C;

       IF A < 0 THEN DO; B=1; C=0; END;

       END ALPHA;
```

In the example above, any of the single statements -- except the DO and END

statements -- is an example of the second form of a group.

Blocks

A block is a collection of statements that defines the program region -- or scope -- throughout which an identifier is established as a name. It also is used for control purposes.

There are two kinds of blocks, begin blocks and procedure blocks.

A begin block has the general form:

```
  [label:] . . . BEGIN-statement
                 program-element-1
                 program-element-2
                         .
                         .
                         .
                 END [label];
```

The label following END must be one of the labels of the BEGIN statement.

A procedure block, or procedure, has the general form:

```
label:  [label:] . . . PROCEDURE-statement
                       program-element-1
                       program-element-2
                               .
                               .
                               .
                       END [label];
```

The label following END must be one of the labels preceding the PROCEDURE statement.

The BEGIN statement and the PROCEDURE statement in the above forms are called heading statements.

While the labels of the BEGIN statement are optional, the PROCEDURE statement must have at least one label.

Although the begin block and the procedure have a physical resemblance and play the same role in delimiting scope of names (see "Scope of Declarations," in this chapter) and defining allocation and freeing of storage (see "Allocation of Data and Storage Classes," in this chapter), they differ in an important functional sense. A begin block, like a single statement, is activated by normal sequential flow, and it can appear wherever a single statement can appear. A procedure can only be activated remotely by CALL statements, by statements in which a CALL option appears or by function references. When a program containing a procedure is executed, control passes around the procedure, from the statement before the procedure statement to the statement after the END statement of the procedure.

Since a procedure can only be activated by a reference to it, every procedure must have a name. The label required for the heading statement of a procedure serves as the procedure name. More than one label provides more than one name.

The procedure name gives a means of activating the procedure at its primary entry point. Secondary entry points can also be defined for a procedure by use of the ENTRY statement. The labels preceding all ENTRY statements and the heading statement of a procedure are collectively called entry names.

As the above definition of block implies, any block A can include another block B, but partial overlap is not possible; block B must be completely included in block A. Such nesting may be specified to any depth.

A procedure that is not included in any other block is called an external procedure. A procedure included in some other block is called an internal procedure.

Every begin block must be included in some other block. Hence, the only external blocks are external procedures.

All of the text of a begin block except the labels preceding the heading statement of the block is said to be contained in the block.

All of the text of a procedure except the entry names of the procedure is said to be contained in the procedure.

That part of the text of a block B that is contained in block B, but not contained in any other block contained in B, is said to be internal to block B.

The entry names of an external procedure are not internal to any procedure and are called external names.

The notion of internal to is vital in the definition of scope (see "Scope of Declarations").

Example:

```
A:  PROCEDURE;
    statement 1
    B:  BEGIN;
        statement 2
        statement 3
        END B;
    statement 4
    C:  PROCEDURE;
        statement 5
    X:  ENTRY;
        D:  BEGIN;
            statement 6
            statement 7
            END D;
        statement 8
        END C;
    statement 9
    END A;
```

In this example, statement i(i=1,...,9) is a labeled or unlabeled simple statement.

As the brackets on the right indicate, block A contains block B and block C, and block C contains block D.

Block A is an external procedure. The procedure name is A, which is an external name, and the only entry name for the procedure.

X is an entry name corresponding to a secondary entry point for procedure C.

Blocks B and D are begin blocks.

Block C is an internal procedure.

The text internal to block A consists of:

```
PROCEDURE;
statement 1
B:
statement 4
C:
X:
statement 9
END A;
```

The text internal to block B consists of:

```
BEGIN;
statement 2
statement 3
END B;
```

The text internal to block C consists of:

```
PROCEDURE;
statement 5
ENTRY;
D:
statement 8
END C;
```

The text internal to block D consists of:

```
BEGIN;
statement 6
statement 7
END D;
```

USE OF THE END STATEMENT

As the examples above imply, the END statement has the form:

END [label];

and is used to terminate a group or a block.

If the optional label following END is not used, the END statement terminates that group or block headed by the DO, BEGIN, or PROCEDURE statement that physically precedes, and appears closest to, the END statement.

If, however, the label L is used following END, the statement terminates that group or block headed by the DO, BEGIN, or PROCEDURE statement <u>with the label L</u> that physically precedes, and appears closest to, the END statement. Any groups or blocks headed by DO, BEGIN, or PROCEDURE statements contained in the terminated block L are also automatically terminated by the END statement END L. This feature eliminates the necessity of writing the intermediate END statements to terminate the contained blocks and groups.

The statement labeled L, which heads the group or block terminated by the END statement END L, is internal to a certain block in the program (see "Blocks," for a definition of <u>internal to</u>). The terminating statement END L, together with its own possible statement-labels, is also considered to be internal to the same block. (If the statement labeled L is a BEGIN or PROCEDURE statement, this block, is of course, the block L.)

The END statement may itself be labeled, and a reference to this label can be made from any part of the program where the label is known. (For a definition of <u>known</u>, see "Basic Rule on Use of Names").

Example:

```
A:  PROCEDURE;       A:  PROCEDURE;
        .                    .
        .                    .
        .                    .
B:  BEGIN;           B:  BEGIN;
        .                    .
        .                    .
        .                    .
A:  PROCEDURE;       A:  PROCEDURE;
        .                    .
        .                    .
        .                    .
C:  DO;              C:  DO;
        .                    .
        .                    .
        .                    .
X:  END B;           END;
END A;               END;
                     X:  END B;
                     END A;
```

In the example on the left above, the statement X:END B terminates the DO group, the internal procedure A, and the block B. The statement END A terminates the external procedure A.

The example on the right is equivalent to the example on the left.

The statement X:END B is internal to block B.

PROCEDURES, FUNCTIONS, AND SUBROUTINES

Formal Parameters

The PROCEDURE statement heading a given procedure and defining the primary entry point to the procedure may specify a list of formal parameters. (For syntax and details of the PROCEDURE statement, see Chapter 6).

One or more ENTRY statements may also be used in the procedure to define secondary entry points. Like the heading statement of the procedure, each of the ENTRY statements must have at least one label to serve as an entry name for that point, and each may specify a list of formal parameters, unrelated to the parameter lists for the other entry points. (For syntax and details see "The ENTRY Statement.")

The formal parameters are identifiers and may appear in statements of the procedure in the context of scalar variable names, array names, structure names, statement label designators, entry names, or file names.

The appearance of an identifier in a formal parameter list for a procedure constitutes a declaration of the identifier as a parameter. This contextual declaration can be combined with an explicit declaration and other contextual declarations in the procedure that will associate required attributes with the parameter. Required attributes not declared explicitly or contextually will be assigned by default.

No declarations of the parameter can appear outside the procedure. (For further details about the restrictions on attributes of parameters see "Arguments and Parameters," in Chapter 8.)

Example:

```
SBPRIM:  PROCEDURE (X, Y, Z);
         DECLARE (X, Y, A, B) FIXED, Z
           FLOAT;
         A = X-1; B = Y+1;
         GO TO COMMON;
SBSEC:   ENTRY (X, Z);
         A = X-2; B = X-3;
         COMMON:  Z = A**2+A*B+B**2;
         END SBPRIM;
```

In this example, the procedure may be entered at its primary entry point SBPRIM, where the formal parameter list is (X, Y, Z), or at its secondary entry point SBSEC, where the formal parameter list is (X, Z).

Procedure References

At any point in a program where an entry name for a given procedure is known, the procedure may be invoked by a procedure reference, which has the form:

```
entry-name [(argument [ ,argument]
   ...)]
```

The number of arguments (possibly zero) in the procedure reference must be equal to the number of formal parameters in the list for the entry point denoted by the entry name.

The procedure invoked by the procedure reference may be an external or an internal procedure. If it is an internal procedure, the block to which the entry name is internal must be active at the time of invocation of the procedure (for a definition of "active," see "Activation and Termination of Blocks," in this chapter).

When a procedure reference invokes a procedure, each argument specified in the reference is associated with its corresponding formal parameter in the list for the denoted entry point, and control is

passed to the procedure at the entry point. The conditions the arguments must satisfy, and the manner of association of each argument with its matching parameter are discussed in "The Arguments in a Procedure Reference."

There are two distinctly different uses for procedures, determined by one of two contexts in which a procedure reference may appear:

1. A procedure reference may appear as an operand in an expression. (For a complete description of expression, see "Expressions," in Chapter 4). In this case, the reference is said to be a function reference, and the procedure is invoked as a function procedure, or simply a function.
2. A procedure reference may appear following the keyword CALL, either in a CALL statement or in a statement using a CALL option. In this case, the reference is said to be a subroutine reference, and the procedure is invoked as a subroutine procedure, or simply a subroutine.

(Ordinarily a given procedure will be used exclusively as a function procedure or exclusively as a subroutine procedure.)

Function References and Function Procedures

When a function reference appears in an expression, the function procedure is invoked. The procedure is then executed, using the arguments, if any, specified in the function reference. The result of this execution is the required value, which is passed with return of control back to the point of invocation. This returned value is then used, in place of the function reference, to evaluate the expression.

The procedure invoked by a function reference normally will terminate execution with a statement of the form RETURN (expression), where expression is a scalar expression of arithmetic, character-string, or bit-string type (see "The RETURN Statement"). It is the value of this expression that will be returned as the function value. The PROCEDURE or ENTRY statement at the invoked entry point may specify data attributes for the function value (see "The PROCEDURE Statement" and "The ENTRY Statement," in Chapter 6). Just prior to return, the expression is evaluated, and, before being passed back, the value is converted, if necessary, to conform to these attributes, or, if the attributes are not specified, to the default attributes implied by the entry name.

The invoked function procedure may also terminate execution on a GO TO statement. In this exceptional case, the evaluation of the expression in which the invoking function reference appeared will be discontinued, and control will go to the designated statement.

Generic Functions

A generic function is a family of functions with a single name. A function reference to a generic function causes the selection of a certain member of the family, depending upon the attributes of the arguments. The characteristics of the value returned depend upon the member that is selected.

Generic functions may be built-in (see below) or specified by the programmer, who may, by means of the attribute GENERIC, define a name to be a generic function name. An entry name may be explicitly declared with the GENERIC attribute. The GENERIC attribute requires a list of all of the entry names of the family and the attributes of all of the arguments for each member (different members must have different argument attribute patterns). Then any function reference appearing in the scope of this declaration and using the declared generic name as an entry name will result in the use of that member of the declared family that has the same argument attribute pattern as the pattern in the argument list of the function reference. For complete details see "Entry Name Attributes," in Chapter 3.

Built-in Functions

Besides function procedures written by the programmer, a function reference may invoke one of a comprehensive set of built-in functions.

The set of built-in functions is an intrinsic part of PL/I. It includes not only the commonly used arithmetic functions but also functions for manipulating strings and arrays, as well as other necessary or useful functions related to special facilities provided in the language. The complete list of these functions and their descriptions can be found in Appendix 1.

A large number of the built-in functions are generic. The built-in generic functions are of considerable convenience to the programmer. He may, for example, always use the same name EXP for the exponential function, regardless of whether the argument is of REAL or COMPLEX mode, regardless of the precision of the argument, etc., and automatically he will

obtain that one of the EXP family that fits the requirements.

Each built-in function, whether or not it is generic, has a specified number of arguments permitted. For some built-in functions only a minimum is specified; additional arguments are optional. For others, a maximum is specified; only one argument is required.

Each of the built-in functions that are not generic has only a single member. When a reference is made to one of these functions, any arguments whose attributes do not match the attributes required by that function are converted to the appropriate form before the function is invoked. The characteristics of the value returned are determined by the function.

Unlike programmer-specified functions, which always return a scalar value, there are many built-in functions that may effectively return an array value when array expressions are used in certain of their argument positions. This facility is useful in combination with the facility of array expressions.

The fixed set of names for the built-in functions is part of the language of PL/I. However, the identifiers corresponding to these names are not reserved; any such identifier can be used by the programmer for other purposes. If the identifier is declared explicitly for some other use, any appearance of the identifier in the scope of this declaration will refer to that other use. The built-in function cannot, of course, be used in this scope. If the identifier appears, but not in the scope of a declaration establishing the identifier for another use, the identifier will be regarded as implicitly declared in the containing external procedure with the attribute BUILTIN, and this appearance will refer to the built-in function.

If an identifier corresponding to a built-in function name is declared with a use other than as the built-in function in some block, the built-in function can be used in contained blocks by declaring the identifier with the attribute BUILTIN.

Subroutine References and Subroutine Procedures

When a procedure is invoked as a subroutine by the execution of a CALL statement or a statement with a CALL option, the initial action is the same as if the procedure were invoked as a function: the arguments in the procedure reference, if any, are associated with the formal parameters and control is passed to the procedure at the denoted entry point. (If the invocation involves a TASK option, the procedure will not necessarily be activated immediately; see "Multi-Task Operations," in this chapter.)

Unlike the function procedure, the subroutine procedure does not return an explicitly specified value to the point of invocation, and control need not necessarily be returned to this point. The procedure may terminate in the following ways:

1. Control reaches a RETURN statement for the procedure. When executed, this statement returns control to the first executable statement logically following the invoking statement, unless the invocation specified a TASK option. If a TASK option has been used, control is simply terminated for this task.
2. Control reaches an END statement for the procedure, which in this case is treated as a RETURN statement. The effect is as in case 1.
3. Control reaches a GO TO statement in the procedure that transfers control out of the procedure. (This is not permitted if the procedure has been invoked by a statement with a CALL option or in a CALL statement with a TASK option.) In this case, control will go to the designated statement (see "The GO TO Statement"). The statement label designator of the GO TO statement may be a parameter of type LABEL, which is associated with a label argument passed from the invoking procedure.
4. Control reaches an EXIT or STOP statement.

Example of Function Reference:

```
COMP:  PROCEDURE;
          .
          .
          .
       S1:  P10=Q5*POLY5(R0, VAL1);
          .
          .
          .
       POLY5:  PROCEDURE (C, X);
               RETURN(C+X*(1+X*(2+X*(3+X*(4
                 +5*X)))));
               END POLY5;
          .
          .
          .
       END COMP;
```

In this example, the external procedure COMP contains the function procedure POLY5, which is invoked when the expression Q5*POLY5(R0, VAL1) is being evaluated dur-

ing execution of the assignment statement labeled S1. When POLY5 is invoked, the arguments R0 and VAL1 will be associated with the parameters C and X, respectively. The returned value for POLY5 (R0, VAL1) will be the value of the expression:

```
R0+VAL1*(1+VAL1*(2+VAL1*(3+VAL1*(4+5*
    VAL1))))
```

Examples of Subroutine Reference:

```
1. COMP:  PROCEDURE;
                .
                .
                .
                .
                .
          S1:  CALL POLY5 (R0, VAL1);
          S2:  P10 = Q5*TEMP;
                .
                .
                .
                .
                .
          POLY5:  PROCEDURE (C, X);
                  TEMP=C+X*(1+X*(2+X*(3+X*
                   (4+5*X))));
                  RETURN;
                  END POLY5;
                .
                .
                .
          END COMP;
```

In the above example, the effect is the same as in the previous example using the function reference. The subroutine procedure POLY5 is invoked by the CALL statement labeled S1. The arguments and parameters are associated as in the previous example, but here, the value of the expression (the same as in the previous example) is assigned within the subroutine to the variable TEMP, which is used by the statement labeled S2, after the RETURN statement passes control back to that statement. Thus, communication of the value is by means of the shared variable TEMP, which, of course, remains available for use following the execution of S2.

In some cases the invoked and the invoking procedure may be separated in such a way that sharing a name in the above simple manner is not possible (see "Scope of Declarations"). Another more general method of communicating values from the invoked procedure, which may be applied in these cases, is illustrated in the following alternative example:

```
2. COMP:  PROCEDURE;
                .
                .
                .
          S1:  CALL POLY5 (R0, VAL1, TEMP);
          S2:  P10=Q5*TEMP;
                .
                .
                .
          POLY5:  PROCEDURE (C,X,Z);
                  Z=C+X*(1+X*(2+X*(3+X*
                   (4+5*X))));
                  RETURN;
                  END POLY5;
                .
                .
                .
          END COMP;
```

Here, the invocation of POLY5 by the CALL statement will associate the variable TEMP with the parameter Z, and the action will be exactly as in the previous example: the parameter Z will effectively be replaced by the name TEMP in the assignment statement for Z, and TEMP will be assigned the value of the expression on the right-hand side, with R0 replacing C and VAL1 replacing X, before return to statement S2. In this case, the value has been communicated from the subroutine through a parameter.

The above two examples illustrate how a single value obtained in a subroutine can be communicated back to the invoking procedure. The action of a subroutine will generally be more complex than this; many communicated variables may be involved, whether scalar, array, structure, or statement-label variables; input/output operations may be specified, etc. In contrast, the usual purpose of a function procedure is to return a scalar value.

The Arguments in a Procedure Reference

An argument in a procedure reference may be any of the following:

1. A scalar expression, an array expression, or a structure expression, where it is understood that formal parameters may appear in the expression in the role of variables
2. A statement-label constant
3. A statement-label variable, including the case of a formal parameter with the LABEL attribute
4. An entry name
5. A formal parameter with the ENTRY attribute
6. A file name
7. A formal parameter with the FILE attribute

The attributes of each argument in a procedure reference must, in general, match the attributes of the corresponding parameter at the named entry point. (An exception in case of a data argument is described below.)

For example, assume that the procedure SUB in a program is defined by:

```
SUB:  PROCEDURE (X, Y, Z);
      DECLARE X FIXED, Y ENTRY, Z LABEL;
      .
      .
      .
      .
      END SUB;
```

This implies that the formal parameter X is used as a fixed-point variable with certain default data attributes, Y is used as an entry name, and Z is a statement label designator in the body of the procedure. Then if SUB is invoked in the program by the statement:

```
CALL SUB (R*S, CALC, L5);
```

it is then necessary that:

1. The expression R*S have all the data attributes of the parameter X (unless SUB is described by an ENTRY attribute, see below).
2. CALC be an entry name.
3. L5 be a statement-label designator.

The Use of the ENTRY Attribute

(The ENTRY attribute is completely described in Chapter 3.)

An identifier is contextually declared to be an entry name in a block if it appears in the block following the keyword CALL or as the function name in a function reference whose argument list is non-empty. If it is desired to use the identifier as an entry name in a block where it is not so declared, the identifier must be given the ENTRY attribute explicitly in a DECLARE statement for the block (see "The DECLARE Statement").

As an illustration, in the above example, the CALL statement:

```
CALL SUB(R*S, CALC, L5);
```

has the entry name CALC as its second argument. This appearance of CALC is not recognizable as an entry name by context, and, unless CALC has been contextually declared as defined above, it must be explicitly defined in the invoking procedure as an entry name by a DECLARE statement. For example:

```
DECLARE CALC ENTRY;
```

A more general form of the ENTRY attribute allows the programmer not only to specify that an identifier is used as an entry name but also to enumerate the attributes of the parameters for the named entry point.

As an illustration, in the above CALL statement example, the three parameters corresponding to the three arguments of the CALL statement might be described in the invoking procedure by the statement:

```
DECLARE SUB ENTRY (FIXED, ENTRY,
   LABEL);
```

This statement specifies that:

1. SUB is an entry name.
2. The entry point SUB has three parameters.
3. The first parameter has the FIXED attribute with certain default data attributes.
4. The second parameter has the ENTRY attribute.
5. The third parameter has the LABEL attribute.

The number of parameters and the attributes of each, as described in the ENTRY attribute specification, must always agree with the number of parameters and their attributes, as defined for the described entry point within the invoked procedure.

One of the applications of the extended form of the ENTRY attribute is mentioned in the immediately following description. (A detailed discussion of the various uses for the ENTRY attribute, including the ABNORMAL, USES, SETS, and GENERIC attributes, can be found in Chapter 3.)

Passing Arguments to the Entry Point

When a procedure is invoked at a given entry point by a procedure reference and each argument is associated with its corresponding formal parameter, the arguments are said to be passed to the entry point.

The action involved in passing the arguments generally will assume that the attributes of each argument match the attributes of its corresponding formal parameter, as described above. However, if the argument is an expression and an explicit ENTRY attribute has been declared for the invoked entry name describing that argument as having different data attributes from the expression as it appears, the expression will be evaluated and converted, before the argument is passed, to conform to the attributes described by the corresponding member of the ENTRY attribute list.

As an illustration, in the preceding example, the first argument in the CALL statement, which invokes the procedure SUB, is the expression R*S. Assume that R*S has

the FLOAT attribute with certain default attributes. These do not match the attributes of the first parameter at the entry point SUB. Then the ENTRY attribute must be used in the invoking procedure to specify the same attributes for the first parameter as specified in the invoked procedure SUB. (The preceding illustration shows one way of doing this.) Thus, on execution of the CALL statement, the expression R*S is evaluated according to the FLOAT attribute and then converted to a fixed-point value with the other required attributes, before being passed to the entry point SUB.

(A detailed description of the action involved in passing arguments to the invoked entry point can be found in Chapter 8.)

In certain circumstances, the preparatory action includes the construction of a dummy argument. For example, a dummy argument is constructed when the argument must be converted, as in the example of R*S just discussed, or when the argument is an expression involving constants or operators (R*S is again an example of this circumstance).

In each of its appearances as a reference in the procedure, the formal parameter corresponding to the argument effectively is replaced by the argument name. Thus, all appearances of the parameter during execution of the procedure are treated as appearances of the argument name. However, in the cases where a dummy argument is constructed, it is the dummy argument name that replaces the parameter. Passing an argument does not always imply a true logical substitution of the argument name for the parameter in the procedure. However, in the important case where the argument is an arithmetic, string, or label variable, a logical substitution does occur. Thus, parameters can be used to communicate values from the invoked procedure back to the invoking procedure. Example 2 of "Subroutine References," above, is an illustration of this.

The Special Procedure Attribute RECURSIVE

In the PROCEDURE statement or ENTRY statement for a given procedure, certain special attributes that characterize the procedure itself may be specified. (For a complete discussion of these attributes, see "The PROCEDURE Statement.") One of these, which has particular significance, is the attribute RECURSIVE. When a procedure in a program is re-activated while it is still active (see "Activation and Termination of Blocks"), the procedure is said to be used recursively. Any procedure used recursively during program execution must be specified with the RECURSIVE attribute.

PREFIXES

A prefix may be attached to a statement.

The prefix is followed by a colon to separate it from the rest of the statement; it precedes the entire statement, including any possible labels for the statement.

The prefix is a list of condition names, separated by commas and enclosed in parentheses; thus a statement with a prefix has the following general form:

```
(condition-name [, condition-
   name] ...) : [label:] ...  statement
```

The condition names are chosen from the following fixed set:

UNDERFLOW
OVERFLOW
ZERODIVIDE
FIXEDOVERFLOW
CONVERSION
SIZE
SUBSCRIPTRANGE
CHECK (identifier list)

NOTE: CHECK (identifier list) may be used as a prefix with the PROCEDURE and BEGIN statements only.

The meanings of these conditions are explained in "The ON Statement," in Chapter 6.

Any of these condition names may be preceded by the word NO; for example, NO CONVERSION can be specified in the prefix list.

Purpose of the Prefix

The conditions named in the prefix to a statement may occur during program execution of a statement lying in the scope of the prefix (see below). If one of these conditions actually does occur, the appearance in the prefix of the corresponding condition name -- or its negation with the word NO -- determines whether or not an interrupt operation will then take place.

Any condition whose occurrence will cause an interrupt is said to be enabled. Enabling of the first five conditions listed above, namely, UNDERFLOW, OVERFLOW, ZE-

RODIVIDE, FIXEDOVERFLOW, and CONVERSION, is provided automatically by PL/I; any occurrence of one of these conditions will cause an interrupt unless the enabling has been negated through the use of a prefix containing the condition name preceded by the word NO. The programmer must himself enable the other conditions through the use of a prefix. For example, no interrupt will occur for a SIZE error (see SIZE condition in Appendix 3), unless the error occurs in a calculation within the scope of a SIZE prefix. For complete details, see "Interrupt Operations" and "The ON Statement."

Scope of the Prefix

The scope of the prefix depends upon the statement to which it is attached.

If the statement is a PROCEDURE or BEGIN statement, the scope of the prefix is the block or group defined by this statement, including all nested blocks, except those for which the condition is re-specified. The scope does not include procedures that lie outside the scope as defined above but which may be invoked by the execution of statements in this scope.

If the statement is an IF statement or an ON statement, the scope of the prefix does not include the blocks or groups that are part of the statement. Any such block may also have an attached prefix, whose scope rules are implied by the other rules given here.

For any other statement, the scope of the prefix is that of the statement itself, including any expressions appearing in the statement but not any procedure explicitly called by the statement.

PROGRAMS

A program is a set of external procedures. Thus, by definition, a program is a set of procedure blocks, each of which is completely nested, and independent of the others.

ATTRIBUTES AND DECLARATIONS

Attributes

An identifier appearing in a PL/I program may refer to one of many classes of objects. It may, for example, represent a variable referring to a complex number expressed in fixed-point form with decimal base; it may refer to a file; it may represent a variable referring to a character string; it may be a statement label or represent a variable referring to a statement label, etc.

Those properties that characterize the object represented by the identifier, and other properties of the identifier itself (such as scope, storage class, etc.), together make up the set of attributes which can be associated with an identifier.

There are a number of classes of attributes. These classes and the attributes in each class are described in Chapter 3.

When an identifier is used in a given context in a program, attributes from certain of these attribute-classes must be known in order to assign a unique meaning to the program. For example, if an identifier is used as a data variable, the data type must be known; if the data type is arithmetic, the base, scale, mode, and precision must be known.

Examples of Attributes:

CHARACTER (50) Association of this attribute with an identifier defines the identifier as representing a variable referring to a string 50 characters in length.

FLOAT Association of this attribute with an identifier defines the identifier as representing a variable referring to arithmetic data, where the data is represented internally in floating-point form.

EXTERNAL Association of this attribute with an identifier defines the identifier as a name with a certain special scope.

Declarations

A given identifier is established as a name, which holds throughout a certain scope in the program (see "Scope of Declarations" in this chapter), and a set of attributes may be associated with the identifier by means of a declaration.

If a declaration is internal to a certain block, then the declared identifier is said to be declared in that block.

In a given program, an identifier may represent more than one name. In this case, each different name represented by the identifier is said to be a different use of the identifier. For example, an identifier may represent an arithmetic variable in one part of a program and an entry name in another part. These two parts, of course, cannot overlap.

Each different use of the identifier is established by a different declaration. References to different uses are distinguished by the rules of scope (see "Scope of Declarations").

Declarations may be explicit, contextual, or implicit.

Explicit Declarations

There are two kinds of explicit declarations:

1. Explicit declarations may be made through use of the DECLARE statement, by which an identifier can be established as a name and given a certain set (possibly empty) of attributes.

   Only one DECLARE statement can be used to establish a given use of a given identifier, and all of the explicitly declared attributes for this use must be specified in the DECLARE statement (for the syntax and use of the DECLARE statement, see "DECLARE Statements," in Chapter 3).

   Example:

```
B:  BEGIN;
    DECLARE (X,V) FLOAT;
    X=R-4 + S ** 2;
    D=X **2 - V**2 + SQRT (X-V + 1);
    END B;
```

   In this example, the identifiers X and V are declared in block B, each with the attribute FLOAT.

2. An identifier may appear as a label of a statement, i.e., as a statement label or an entry name.

   In this case, the label or name is said to be declared in the block to which it is internal (for the definition of internal to, see "Blocks"). This implies that every statement label except the label of a BEGIN statement is declared in the block to which its associated statement is internal. It further implies that a label appearing before a BEGIN, ENTRY, or PROCEDURE statement is declared in the immediately containing block.

   In the special case where the label is an entry name for an external procedure, the name is said to be declared externally, and has the EXTERNAL attribute (see "Scope of Declarations").

   Example:

```
A:  PROCEDURE;
        .
        .
        .
    P:  PROCEDURE;
        .
        .
        .
        LOOP:DO I=1 TO N;
             .
             .
             .
           Q:  BEGIN;
               LOOP:DO J=0 TO I;
                    .
                    .
                    .
                    END LOOP;
               END Q;
           END LOOP;
        END P;
    END A;
```

   In this example:
   A is declared as an external entry name.
   P is declared as an entry name in block A.
   LOOP in its first use is declared as a statement label in block P.
   Q is declared as a statement label in block P.
   LOOP in its second use is declared as a statement label in block Q.

Contextual Declarations

The syntax of PL/I allows identifiers appearing in certain contexts to be recognized without an explicit declaration. The various cases are described below.

1. An identifier may occur in a context where only a file name may appear. In this case, the identifier is said to be declared as a file name, with the default attribute EXTERNAL (see "Default Attributes," in Chapter 3.

   Example:

```
        READ FILE (INFILE) DATA;
```

Here, INFILE is declared contextually with attribute FILE.

2. An identifier may occur in a context where only a task identifier (see "The TASK Option," in Chapter 6. may appear. In this case, the identifier is said to be declared as a task identifier, with the default attribute EXTERNAL.

Example:

```
WAIT (TASK2);
```

Here, TASK2 is declared contextually as a task identifier.

3. An identifier may occur in a context where only a programmer-specified condition name (see Appendix 3) may appear. In this case, the identifier is said to be declared as a condition name, with the default attribute EXTERNAL.

Example:

```
ON CONDITION (TEST1) GO TO CHECK;
```

Here, TEST1 is declared contextually as a condition name.

4. An identifier may occur within a statement in a context where only an entry name may appear. In this case, the identifier is said to be declared as an entry name. If the occurrence of the identifier does not lie within the scope of the same identifier used to label a PROCEDURE or ENTRY statement, the identifier is given a default attribute of EXTERNAL.

Example:

```
CALL EXPRI;
```

5. An identifier may appear in a formal parameter list in a PROCEDURE or ENTRY statement. In this case, the identifier is said to be declared in the block to which the list is internal. Attributes may be explicitly declared for the identifier in a DECLARE statement internal to the same block, in which case both the contextual and explicit declarations are regarded as constituting a single declaration.

Example:

```
PAY:  PROCEDURE (HOURS, RATE);
      DECLARE HOURS FIXED (6,2);
      .
      .
      .
      END PAY;
```

In this example, HOURS is declared explicitly and RATE contextually in the block PAY.

Implicit Declarations

An identifier may be used in a procedure but not be explicitly declared or contextually declared. In this case the identifier is said to be <u>implicitly declared</u> in the containing external procedure. As will be seen in the discussion of scope, this implicit declaration will then apply to the entire external procedure block except for any contained blocks where the identifier might be re-declared.

Example:

```
B1:    PROCEDURE (Z1,Z2);
       TEMP1=ABS (Z1*2+Z2**2);
       B2:  BEGIN;
            TEMP2= 1/(TEMP1+Z2)**2;
       IF TEMP2>TEMP1 THEN RETURN
             (TEMP2);
       END B2;
       RETURN (TEMP1);
       END B1;
```

In this example, TEMP1 and TEMP2 are both implicitly declared in block B1.

The IMPLICIT Statement and Default Attributes

The declaration of a given identifier in a program need not necessarily specify all, or even any, of the attributes of the identifier which are required to be known.

The IMPLICIT statement (for syntax and details, see "IMPLICIT Statements," in Chapter 3) can be used to associate attributes with an identifier , where the attributes are required to be known but are not explicitly or contextually declared. For example, an IMPLICIT statement, which holds for the entire external procedure in which it appears, might specify that any identifier commencing with the letter Z is to have the attribute COMPLEX, whenever the identifier is used as an arithmetic variable and its mode (REAL or COMPLEX) is not declared.

For any attribute required for a given identifier but not declared either explicitly or contextually, there may exist an IMPLICIT statement in the program which implies the missing attribute. If the missing attribute is not supplied by the IMPLICIT statement, there exists a complete set of rules in the language that assigns the attribute <u>by default</u>. In general, these rules supply those attributes that would be the most probable in the particular context. The rules can be found in Chapter 3.

SCOPE OF DECLARATIONS

When a declaration of an identifier is made in a program, there is a certain well-defined region of the program over which this declaration is applicable. This region is called the *scope of the declaration* or the *scope of the name established by the declaration*.

The scope of a declaration of an identifier is defined as that block B to which the declaration is internal, but excluding from block B all contained blocks to which another declaration of the same identifier is internal.

This definition of scope can be applied to all identifier declarations except the declaration of entry names of external procedures (see "Declarations," in this chapter). The appearance of an identifier as the entry name of an external procedure is regarded as an explicit declaration of the identifier as an entry name with the EXTERNAL attribute. The scope of such a declaration is defined to be the entire external procedure, excluding all contained blocks to which another declaration of the same identifier is internal.

Scope of External Names

In general, distinct declarations of the same identifier imply distinct names with distinct non-overlapping scopes. It is possible, however, to establish the same name for distinct declarations of the same identifier by means of the EXTERNAL attribute. The EXTERNAL attribute is defined as follows:

> An explicit or contextual declaration of an identifier that declares the identifier as EXTERNAL is called an *external declaration for the identifier*. All external declarations for the same identifier in a program will be linked and considered as establishing the same name. The scope of this name will be the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must be consistent, since the declarations all involve a single name. For example, it would be an error if the identifier ID were used as a file name in some READ statement in a program, and in the same program to declare ID as EXTERNAL ENTRY, since a file name always has the scope attribute EXTERNAL (see "Default Attributes," in Chapter 4) and the attribute FILE, which conflicts with the attribute ENTRY.

The EXTERNAL attribute can be used to communicate between different external procedures or to obtain non-continuous scopes for a name within an external procedure.

An *external name* is a name that has the scope attribute EXTERNAL. If a name is not external, it is said to be an *internal name* and has the scope attribute INTERNAL.

Example 1:

```
1   A:  PROCEDURE;
2       DECLARE (X,Z) FLOAT;
             .
             .
             .
3       B:  PROCEDURE (Y);
4           DECLARE Y BIT (6);
5           C:  BEGIN;
6               DECLARE (A,X) FIXED;
                    .
                    .
                    .
7               Y:  RETURN;
                END C;
            END B;
8       D:  PROCEDURE;
9           DECLARE X FILE;
10          Y = Z;
                .
                .
                .
            END D;
        END A;
```

The numbers on the left are for reference only, and are not part of the procedure. See Table 1 for an explanation for the scope and use of each name.

Since entry names of external procedures and file names have attribute EXTERNAL, the scope of the entry name A and of the file name X above may include parts of other external procedures of the program.

Table 1. Scope and Use of Names in Example 1, for "Scope of External Names"

Reference Line	Name	Use	Scope (by block names)
1	A	external entry name	all of A except C
2	X	floating-point variable	all of A except C and D
2	Z	floating-point variable	all of A
3	B	internal entry name	all of A
4	Y	bit string	all of B except C
5	C	statement label	all of B
6	A	fixed-point variable	all of C
6	X	fixed-point variable	all of C
7	Y	statement label	all of C
8	D	internal entry name	all of A
9	X	file name	all of D
10	Y	floating-point variable	all of A except B

Example 2:

```
        A:  PROCEDURE;
1           DECLARE X EXTERNAL;
                .
                .
                .
            B:  PROCEDURE;
2               DECLARE X FIXED;
                    .
                    .
                    .
                C:  BEGIN;
3                   DECLARE X EXTERNAL;
                        .
                        .
                        .
                    END C;
                END B;
            END A;
        D:  PROCEDURE;
4           DECLARE X FIXED;
                .
                .
                .
            E:  PROCEDURE;
5               DELCARE X EXTERNAL;
                    .
                    .
                    .
                END E;
            END D;
```

The reference numbers on the left are not part of the procedure.

In example 2, there are five declarations for the identifier X.

Declaration 2 declares X as a fixed-point variable name; its scope is all of block B except block C.

Declaration 4 declares X as another fixed-point variable name, distinct from that of declaration 2; its scope is all of block D except block E.

Declarations 1,3,5 all establish X as a single name; its scope is all of the program except the scopes of declarations 2 and 4.

Basic Rule on Use of Names

A name is said to be <u>known</u> only within its scope. This definition suggests a basic -- and almost self-evident -- rule on the use of names:

<u>All appearances of a given name in a program must lie within the scope of that name</u>.

There are many implications to the above rule. One of the most important is the limitation of transfer of control by the statement GO TO A, where A is a statement label.

The statement GO TO A, internal to a block B, can cause a transfer of control to another statement internal to block B or to a statement in a block containing B, and to no other statement. In particular, it cannot transfer control to any point within a block contained in B.

DYNAMIC PROGRAM STRUCTURE

PROGRAM CONTROL

Every program, when it is being executed, has a <u>control</u> that determines the order of execution of the statements. For a discussion of their order see "Sequence of Control," in Chapter 6.

Execution of the program is initiated by the operating system, which invokes the initial procedure. This initial procedure must be an external procedure that has been specified with the MAIN attribute (see "The PROCEDURE Statement," in Chapter 6). This procedure cannot have CONTROLLED parameters.

ACTIVATION AND TERMINATION OF BLOCKS

A begin block is said to be <u>activated</u> when control passes through the BEGIN statement for the block. A procedure block is said to be <u>activated</u> when the procedure is invoked at any one of its entry points.

During certain time intervals of the execution of a program, a block may be <u>active</u>. A block is active if it has been activated and is not yet <u>terminated</u>.

There are a number of ways in which a block may be terminated. These are implied by the following rules:

1. A begin block is terminated when control passes through the END statement for the block.
2. A procedure block is terminated on execution of a RETURN statement or an END statement for the block. (The END statement implies a RETURN statement; see Chapter 6.)
3. A block is terminated on execution of a GO TO statement contained in the block which transfers control to a point not contained in the block.
4. The execution of a STOP statement causes all blocks in the program to be terminated.
5. The execution of an EXIT statement causes termination of the task containing the statement and all tasks attached by this task. Thus, all blocks corresponding to these tasks are terminated.
6. When a block B is terminated, all of the dynamic descendants of B also are terminated.

Dynamic Descendance

If a block B is activated and control stays at points internal to B until B is terminated, no other blocks can be activated while B is active. (This discussion is not applicable to the multi-task, or asynchronous, mode of operation, which implies more than a single control; see "Multi-Task Operation.")

However, another block, B1, may be activated from a point internal to block B while B still remains active. This is possible only in the following cases:

1. B1 is a procedure block immediately contained in B (the label of B1 is internal to B) and reached through a procedure reference.
2. B1 is a begin block immediately contained in B and reached through normal flow.
3. B1 is a procedure block not contained in B and reached through a procedure reference. (B1, in this case, may be identical to B, but conceptually it is to be regarded dynamically as a different block.)
4. B1 is a begin block or a statement specified by an ON statement (see "The ON Statement"), and reached through an interrupt. (For present purposes, even if B1 is a statement, it can be regarded as a block, and this case is dynamically similar to case 1 or case 3 above.)

In any of the above cases, B1 is said to be an <u>immediate dynamic descendant</u> of B, while B1 is active.

Block B1 may itself have an immediate dynamic descendant B2, etc., so that a chain of blocks (B, B1, B2,...) is created, where, by definition, all of the blocks are active. In this chain, each of the blocks B1, B2, etc., is said to be a <u>dynamic descendant of B</u>.

An implication of rule 6 for the termination of blocks is the following rule: When block B is terminated, all active blocks contained in block B also are terminated.

It is important for the programmer to note that the termination of a given block may automatically imply the termination of other blocks and that these blocks need not necessarily be contained in the given block; storage for all AUTOMATIC variables declared in these blocks will be released at the time of termination.

Dynamic Encompassing

Block A dynamically encompasses block B, or block B is dynamically encompassed by block A, if B is a dynamic descendant of A.

ALLOCATION OF DATA AND STORAGE CLASSES

Because the internal storage of any computer is limited in size, the efficient use of this storage during the execution of a program is frequently a crucial consideration. The simple static process of data allocation used by many compilers -- the assignment of a distinct storage region for each distinct variable used in the source program -- may be wasteful. Multiple use of a storage region for different data during program execution can reduce the total amount of storage required.

Provisions are included in the language to give the programmer virtually any degree of control over the allocation of storage for the data variables in a program. On the other hand, the entire problem of allocation can be ignored completely by the programmer, if storage economization is of little significance in his situation, and a reasonably efficient use of storage usually will still be obtained automatically.

Definitions and Rules

Storage is said to be allocated for a variable when a certain region of storage is associated with, or assigned to, the variable. Allocation for a given variable may take place statically, before execution of the program, or dynamically, during execution.

Storage may be allocated dynamically for a variable and subsequently released. Thus, this storage is freed for possible use in later allocations. If storage has been allocated for a variable and not subsequently released, the variable is said to be in an allocated state.

When a variable appears in a statement of a source program, the appearance is called a reference if it corresponds either to the assignment of a value to the variable (e.g., an appearance on the left side of an assignment statement) or to a use of the value of the variable (e.g., appearance in an expression to be evaluated).

At any point where a variable appears as a reference, it must be in an allocated state.

Storage Classes

Every variable in a program, except variables with the DEFINED attribute (see Chapter 3), must have a storage class, which specifies the manner of storage allocation for the variable.

There are three storage classes. The storage class is specified by declaring the variable with one of the three storage class attributes STATIC, AUTOMATIC, or CONTROLLED. The storage class may be declared explicitly, contextually, or by default.

The Static Storage Class

Storage for a variable with attribute STATIC is allocated before execution of the program and is never released during execution.

The scope attribute (see Chapter 3) of a STATIC variable may be INTERNAL or EXTERNAL. An EXTERNAL variable with unspecified storage class has, by default, the STATIC storage class attribute.

The Automatic Storage Class

If a variable has the attribute AUTOMATIC, the block in which this variable is declared (see "Attributes and Declarations") determines dynamic allocation for the variable. Whenever this block is activated during execution of a program, storage will be allocated for the variable, and the variable will remain in an allocated state until termination of the block. At the time of termination, the storage will be released. Thus, the time interval during which the variable is known (see "Scope of Declarations").

Termination of a block by means of a GO TO statement may imply simultaneous termination of other blocks and, consequently, simultaneous release of storage for all AUTOMATIC variables declared in these blocks (see "The GO TO Statement").

If the block is a procedure and is called recursively (reactivated one or more times before return), previously allocated storage for the AUTOMATIC variable is pushed down on each entrance and popped up on each return to yield the proper generation of storage for the variable after each return, until the final return out of the procedure.

Note: The terms "pushed down" and "popped up" refer to the notion of a push-down stack. A push-down stack is a logical device S, similar in behavior to a physical stacking process. When an element is placed in S, it is conceptually placed on top of the elements already in S, which are "pushed down." At any time, if S is not empty, the top element -- the element most recently placed in S -- can be removed from S, and the remaining elements are "popped up."

The scope attribute (see Chapter 3) of an AUTOMATIC variable must be INTERNAL. An INTERNAL variable with unspecified storage class has, by default, AUTOMATIC storage class attribute.

The Controlled Storage Class

If a variable has the attribute CONTROLLED, storage allocation must be explicitly specified for the variable by the ALLOCATE and FREE statements.

The ALLOCATE statement (see Chapter 6) specifies one or more variables, each with certain optional attributes. Execution of the statement causes the allocation of storage for the variable specified.

The FREE statement specifies one or more variables, and execution of the statement causes the storage most recently allocated for the variables to be released.

At some point in a program, it may not be known whether a variable X is in an allocated state. The built-in function ALLOCATION (see Appendix 1) is provided to test this state. The function reference ALLOCATION (X) will return the value binary 1 if X is in an allocated state, and the value is binary 0 if not.

The scope attribute of a CONTROLLED variable may be INTERNAL or EXTERNAL.

Example:

```
A:  PROCEDURE;
    DECLARE X STATIC;
        .
        .
        .
    B:  PROCEDURE;
        DECLARE Y (100) CONTROLLED, Z CHAR-
            ACTER (1000);
            .
            .
            .
        ALLOCATE Y;
            .
            .
            .
        FREE Y;
            .
            .
            .
        C:  BEGIN;
            DECLARE Z (100);
              .
              .
              .
            END C;
            .
            .
            .
        RETURN;
            .
            .
            .
        END B;
         .
         .
         .
    END A;
```

Assume in the above example that the termination of procedure A occurs on the return implied by END A, the termination of procedure B occurs on the RETURN statement, and the termination of block C occurs at END C. Then in this example:

Storage for the static variable X is allocated before execution and is never released.

The character-string variable Z is AUTOMATIC by default. Storage is allocated for this Z on entrance to procedure B and is released on execution of the RETURN statement.

The array-variable Z is AUTOMATIC by default. Storage is allocated for this Z at the beginning of execution of block C and is released at END C.

Storage for the CONTROLLED variable Y is allocated on execution of the ALLOCATE statement and is released on execution of the FREE statement. After execution of the FREE statement, the variable Y presumably is not used, but the

character-string variable Z can be used, since storage is not released for this variable until the termination of procedure B.

MULTI-TASK OPERATION

In most computer jobs, it is by no means inherently necessary that all of the steps required in the job be carried out in a serial time order. The job could frequently be so arranged that part or all of several of its required sub-jobs might be carried out simultaneously; in fact, many jobs are often naturally structured in this way.

The notion of *task*, together with several related facilities, has been introduced in PL/I chiefly to enable the programmer to exploit these multi-processing capabilities, in the frequent applications where this is possible.

Definition and Characteristics of a Task

During program execution, a procedure may be invoked by a CALL statement that specifies a TASK option (see "The CALL Statement" and "The TASK Option," in Chapter 6, for syntax). The specification of this option includes an arbitrary *task identifier*.

When a CALL statement using a TASK option is executed, the *execution of the invoked procedure* (including the execution of any procedures invoked by it) is said to be *assigned as a task*. The *name of the task* is the specified task identifier.

Certain statements (see READ, WRITE, OPEN, CLOSE, SORT, FETCH and DISPLAY *statements*) may also be specified with a TASK option. If the TASK option is specified for any one of these statements, the *execution of the statement itself* (including the execution of any procedures whose invocation may be implied by the execution of the statement) is also said to be *assigned as a task*.

Thus, a *task* is a dynamic sequence of activities that is required to be carried out (in contrast to the static sequence of statements, e.g., a procedure, which *defines* this sequence of activities) and whose execution is assigned as a task.

Every task, during the time interval it actually is being carried out, requires a control mechanism. Although, of course, only one task can be carried out at a time for a control mechanism, it is possible to operate as if one were in a multi-task environment by switching the single control from one task to another as the timing requires.

When, during the activities of task A, the control for task A reaches a CALL statement with a TASK option (or one of the other possible statements with a TASK option), the specified task, say task B, is said to be *attached* to task A, with A as the *attaching* task. (Task B is, by definition, a part of task A.)

When task B is attached, it may or may not be carried out immediately; the decision will be made by an implementation-dependent scheduler, which is part of the local operating system and which schedules tasks on the basis of the currently available resources and the relative priorities assigned to the tasks either by the programmer or by the system. Under ideal circumstances, when task A attaches task B, task B commences immediately and task A continues past the point of attachment; thus the tasks operate concurrently, each independently, with its own control.

Whatever decision is made by the scheduler when task A attaches task B, the *operation* of task A and task B is said to be *asynchronous*, and the procedure (or statement) defining task B is said to be *invoked asynchronously*.

The time relationships among tasks may be schematized by a simple chart, as for example:

```
              C-----
              |
          B---+----------
          |
          |         D----------
          |         |
     A----+---------+---------------
```

In this example, a horizontal increment represents a unit of time, a horizontal line segment on the left denotes a task, and a vertical segment denotes an attachment of the upper task to the lower task at that point in time. (Here, task A has attached task B and later, task D; meanwhile, B has attached C.)

Termination of Tasks

A task may be terminated in one of the four following ways:

1. Control for the task reaches an EXIT

statement (see Chapter 6 for a discussion of each of the statements mentioned here)

2. Control for any task reaches a STOP statement
3. Control for the task reaches a RETURN statement for the procedure defining the task
4. Control for the task reaches an END statement for the procedure defining the task

It should be noted that when a task A is terminated, any tasks attached to task A, being part of task A, also are terminated.

Synchronization of Tasks

When task A attaches task B, even though task A and task B operate asynchronously, task A always has a direct means of synchronizing task B and itself by use of the WAIT statement. In its simplest form, an example of this statement is:

```
WAIT (B);
```

If WAIT (B) is executed in task A at any time after task B is attached to task A, the operation of task A will be delayed until task B is terminated.

As a simple schematic example, suppose the task defined by procedure P invokes procedure Q asynchronously:

```
P:  PROCEDURE;
    .
    .
    .
    CALL Q (A1, A2, A3), TASK (B);
    statement 1
    .
    .
    .
    WAIT (B);
    statement 2
    .
    .
    .
    END P;
```

When the CALL statement is executed, task B (defined by procedure Q) is attached, and execution of Q may proceed asynchronously with the execution of statement 1 of procedure P, etc., up to the WAIT statement. If task B has been terminated before the WAIT statement is executed, execution in procedure P will continue immediately with statement 2; otherwise, the execution of P will be delayed until task B has been terminated. Thus, the programmer can be certain that at the time statement 2 is executed, all of the activities of task B have been completed.

An additional facility that task A may use after it has attached task B is the built-in function COMPLETE, which allows task A to interrogate the termination status of task B. When the function reference, COMPLETE (B), is used in task A, the value returned will be '1'B if task B has been terminated, and '0'B if it has not.

Both the WAIT statement and the COMPLETE function can be applied by the attaching task only to an immediately attached task. (Thus, if task A attaches task B which in turn attaches task C, task A can wait on or test B, but not C; and B can wait on or test C, but not A.)

Allocation of Data in Tasks

The rules of scope and storage allocation hold across task boundaries. If storage is allocated for a variable in the attaching task, this allocation may apply to the attached task, so that the variable may appear as a reference in the attached task. It is the responsibility of the programmer to be certain that storage for such a variable is not released too early in the attaching task. (Normally, this is done by synchronizing by use of the WAIT statement.)

(Further details concerning tasks as related to storage allocation and other special considerations can be found in Chapter 8; also see "The WAIT Statement" for other information and examples.)

INTERRUPT OPERATIONS

During the course of program execution any one of a certain set of conditions may occur that can result in an <u>interrupt</u>. An interrupt operation causes the suspension of normal program activities, in order to perform a special action; after the special action, program activities may or may not resume at the point where they were suspended. The time point of an interrupt is, in general, unpredictable.

For most conditions that can cause an interrupt, the special action to be taken may be specified by the programmer. To do this, he may specify the condition in an ON statement; therefore these conditions are known as the ON-conditions. A complete list and description of the ON-conditions can be found in Appendix 3. With two exceptions (see "Programmer Defined ON-Conditions," in this chapter), each ON-

condition is named with a unique identifier suggestive of the condition (e.g., ZERODIVIDE names the condition obtaining whenever an attempt is made to divide by zero). This collection of names, like the built-in function names, is an intrinsic part of the language, but the names are not reserved; the programmer may use them for other purposes, so long as no ambiguity exists.

Use of the ON Statement

In order to define the action to be taken when an interrupt occurs, the programmer may write an ON statement, which has the general form:

ON condition-specification action-specification

The "condition specification" either is an ON-condition name or denotes a programmer-defined condition, and the "action specification" is a single statement or begin block, optionally preceded by the keyword SNAP (see "The ON Statement" for complete syntax and details).

When an ON statement that is internal to a given block (for example, a block B) is executed, it causes a preparatory action with the following effect:

> If, during the execution of any statement after the execution of the ON statement and before the termination of block B (including the execution of statements in all dynamic descendants of block B), the condition specified in the ON statement ever occurs and an interrupt results, the statement or begin block specified in the ON statement will be executed as though it were invoked as a procedure block. (If SNAP also has been specified, a standard action will precede this pseudo-invocation.) Control normally will be returned to the point following the interrupt.

When an ON statement specifying a given condition is executed, the action to be taken is established by the execution. The time interval during which this action specification is effective is defined above in the description of the effect of an ON statement. There are two qualifications to this description:

1. If, after a given action is established by execution of an ON statement, and while this action specification is still effective, another ON statement specifying the same condition is executed, then this latter ON statement will take effect as described above, so that its specified action will determine the interrupt action for the given condition. (The effect of the old ON statement is either temporarily suspended or completely nullified, depending upon whether or not the new ON statement is in a block dynamically descendant from the block to which the old ON statement is internal; see "The ON Statement" and "The REVERT Statement" for more details.)
2. There are eight ON-conditions whose names (possibly preceded by the word "NO") may appear in a prefix to a statement (see "Prefixes," in this chapter). Even when one of these conditions appears in an ON statement, occurrence of the condition will not necessarily result in an interrupt. For an interrupt to occur, there are certain additional requirements, which are described in the following paragraph.

   There are three of these eight ON-conditions, SIZE, SUBSCRIPTRANGE, and CHECK (identifier list), for which an interrupt will not take place when the condition occurs unless the programmer specifically designates that the interrupt _is_ to take place. He may enable this condition by explicitly specifying the condition in a prefix whose scope will cover the calculation where the condition may occur. If a calculation resulting in the occurrence of either of these conditions does not lie within the scope of such a prefix, no interrupts will occur. The other five of these eight special ON-conditions, namely OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and FIXEDOVERFLOW, are always enabled, but the programmer may specifically designate that an interrupt is _not_ to take place. An interrupt for any one of these conditions will always take place when the condition occurs unless the occurrence is in a calculation lying within the scope of a prefix specifying NO OVERFLOW, NO UNDERFLOW, NO ZERODIVIDE, NO CONVERSION, or NO FIXEDOVERFLOW, respectively.

All other conditions, whose names cannot be used in a prefix, are always enabled.

System Interrupt Action

With the exception of the two programmer-defined ON-conditions, each of the ON-conditions has a standard action

statements whose label appears in the list is executed, the condition defined by this specification is regarded as occurring, and an interrupt will take place. (For a precise explanation of this kind of condition, see Appendix 3, "ON Conditions.")

Facilities for Program Checkout

The programmer-specified condition described above is a powerful tool for program checkout. As an example of its use, suppose that the following statement is executed:

```
ON CHECK (A, SUB1, ST5) block
```

In the example, A is a data variable, SUB1 is a procedure name, and ST5 is a statement label. Then, whenever a value is assigned to A (or to any part of A, if A is an array or structure name), an interrupt occurs, and the arbitrary block specified in the ON statement is executed. Similarly, whenever the procedure SUB1 is invoked or the statement labeled ST5 is executed, an interrupt occurs. The interrupt action defined by the block usually will involve the recording of some information; if desired, a complete history of A's assigned values or of the executions of SUB1 or ST5 may be kept.

Another useful ON-condition is the condition named SUBSCRIPTRANGE. Parts of the program can be designated by the programmer, using the keyword SUBSCRIPTRANGE in appropriate prefixes, to receive constant monitoring of subscript values. Whenever the value of some subscript in some array goes out of its designated range, an interrupt will occur, and action, specified by a previously executed ON statement, will take place to correct the error.

The SIGNAL statement also will be found useful for checkout, since it can be used to simulate the occurrence of any ON-condition (see "The SIGNAL Statement").

DATA TYPES

Information that is operated on in a PL/I object program during execution is called data. Each data item has a definite type and representation.

The permitted data types are arithmetic, character string, bit string, and label. The details for the specification of data type attributes are contained in the sections "DECLARE Statements" and "The Attributes," in Chapter 3.

ARITHMETIC DATA

An arithmetic data item is one that has a numeric value with characteristics of base, scale, mode, and precision. The data item may be represented either as a numeric field or in a coded form, that is, in an internal representation that is implementation dependent. A numeric field is a string of characters that is given a numeric interpretation by means of the PICTURE attribute (see Chapter 3). The base, scale, and precision are all specified in the picture of the numeric field. A data item in coded form does not have a PICTURE attribute, but has its characteristics given by the attributes specifying base, scale, mode, and precision.

Base (decimal or binary), scale (fixed-point or floating-point), and precision have reference to internal representation of the data described and to the internal arithmetic that is to be used.

Base

Arithmetic data may be specified as having either decimal or binary base.

Scale

Arithmetic data may be specified as having either fixed-point or floating-point scale. Fixed-point data items are rational numbers for which the number of decimal or binary digits are specified; the position of the decimal or binary point may also be specified. Floating-point data items are rational numbers in the form of a fractional part and an exponent part.

Mode

Arithmetic data may be operated on in either the real or complex mode. In the complex mode, a data item is considered to consist of a number pair, the first member of the pair representing the real part of the complex number and the second, the imaginary part.

Precision

The precision of fixed-point data (w,d) is specified by giving the total number of binary or decimal digits, w, to be maintained and a scale factor, d. The precision of floating-point data is specified by giving only the total number of binary or decimal digits to be maintained, w.

CHARACTER-STRING DATA

Character-string data consists of a string of zero or more characters in the data character set (see "Data Character Set," in Chapter 1). The string may be fixed or varying in length. The actual number of characters must be specified if it is of fixed length, and the maximum length must be specified if it is of varying length.

BIT-STRING DATA

Bit-string data consists of a string of zero or more binary digits (0 and 1). The string may be fixed or varying in length. The actual length of the field must be specified if it is of fixed length, and the maximum length must be specified if it is of varying length.

ARRAYS

An array is an n-dimensional, ordered collection of elements, all of which have identical data declaration. If arithmetic, all of the elements of the array must have the same base, scale, mode, and precision or the same picture. If character-string or bit-string, all of the elements must have the same actual length if fixed length, and the same maximum length if varying length. The number of dimensions of an array, and the upper and lower bounds of each dimension, are specified by the use of the dimension attribute.

Example:

```
DECLARE   A(3,n);
```

This statement defines A as an array with 2 dimensions, 3 rows, and n columns. The matrix given below illustrates the array A.

$$\begin{matrix} A_{11} & A_{12} & A_{13}\ldots A_{1n} \\ A_{21} & A_{22} & A_{23}\ldots A_{2n} \\ A_{31} & A_{32} & A_{33}\ldots A_{3n} \end{matrix}$$

The elements of an array may be structures (see "Arrays of Structures").

STRUCTURES

A structure is a hierarchical collection of scalar variables, arrays, and structures. These need not be of the same data type nor have the same attributes.

Structures may contain structures. The outermost structure is the major structure, and contained structures are minor structures. A major structure must be at level one. Contained structures must always have a level number numerically greater than the structure in which they are contained. The level number may be followed by an optional blank.

Example:

1. DECLARE 1 PAYROLL, 2 NAME, 2 HOURS, 3 REGULAR, 3 OVERTIME, 2 RATE;

   takes the form:

```
1 PAYROLL
   2NAME
   2HOURS
      3REGULAR
      3OVERTIME
   2RATE
```

In the above example PAYROLL is defined as the major structure containing the scalar variables NAME and RATE and the structure HOURS. The structure HOURS contains the scalar variables REGULAR and OVERTIME.

```
DECLARE 1 A, 2 B, 2 C, 3 D (2), 3 E, 2
F;
```

This takes the form:

```
A
    B
    C
        D(1)
        D(2)
        E
    F
```

The decimal integers before the identifiers specify the level; the decimal integer in parentheses specifies the bounds of the one-dimensional array. A is defined as the major structure and contains the minor structure C and the scalar variables B and F. C contains D, a one-dimensional array with two scalar variables, and the scalar variable E.

ARRAYS OF STRUCTURES

An array of structures is formed by giving the dimension attribute to a structure. This dimension attribute causes all contained items to be arrays.

Examples:

1. DECLARE 1 CARDIN(3), 2 NAME, 2 WAGES, 3 NORMAL, 3 OVERTIME;

   The decimal integers before the identifiers specify the level. The name, CARDIN, represents a one-dimensional structure of bounds 1 to 3, that is, CARDIN is an array of structures. Each element of the structure contains the variable, NAME, and the minor structure, WAGES. WAGES contains the variables, NORMAL and OVERTIME. Because CARDIN has a dimension specified, NAME, NORMAL, and OVERTIME are arrays, and their elements are referred to by subscripted names.

   The form of the data is:

```
CARDIN  (1)   NAME   (1)
              WAGES  (1)   NORMAL    (1)
                           OVERTIME  (1)

CARDIN  (2)   NAME   (2)
              WAGES  (2)   NORMAL    (2)
                           OVERTIME  (2)

CARDIN  (3)   NAME   (3)
              WAGES  (3)   NORMAL    (3)
                           OVERTIME  (3)
```

2. DECLARE 1 X, 2 Y, 2 Z (2), 3 P (2:3,2), 3 Q, 2 R;

X is an undimensioned major structure containing scalar variables, arrays, and a structure.
Y is a scalar variable
Z is a dimensioned structure containing P and Q
P is a three-dimensional array
Q is a one-dimensional array
R is a scalar variable

The form of the data is:

```
     ┌ Y
     │          ┌ P (1,2,1)
     │          │ P (1,2,2)
     │  Z (1)   │ P (1,3,1)
     │          │ P (1,3,2)
     │          └ Q (1)
X    │
     │          ┌ P (2,2,1)
     │          │ P (2,2,2)
     │  Z (2)   │ P (2,3,1)
     │          │ P (2,3,2)
     │          └ Q (2)
     └ R
```

NAMING

This section describes the rules for referring to a particular data item, groups of items, arrays, and structures. The permitted types of data names are simple, qualified, subscripted, and subscripted qualified.

SIMPLE NAMES

A simple name is an identifier (see "Identifiers," in Chapter 1) that refers to a scalar, an array, or a structure.

SUBSCRIPTED NAMES

A subscripted name is used to refer to an element of an array. It is a simple name that has been declared to be the name of an array followed by a list of subscripts. The subscripts are separated by commas and are enclosed in parentheses. A subscript is an expression that is evaluated and converted to an integer before use (see "Evaluation of Expressions," in Chapter 4). The number of subscripts must be equal to the number of dimensions of the array, and the value of a specified subscript must fall within the bounds declared for that dimension of the array.

A subscripted name takes the form:

identifier (subscript [, subscript] ...)

Examples:

```
A (3)
FIELD (B,C)
PRODUCT (SCOPE * UNIT + VALUE, PERIOD)
ALPHA (1,2,3,4)
```

Cross Sections of Arrays

The concept of cross sections is a logical extension of the subscripting notation. A cross section of an array is referred to by the array name, followed by a list of subscripts, at least one of which is an asterisk. The subscripts are separated by commas, and the entire list is enclosed in parentheses. The number of items in the list must be equal to the number of dimensions of the array. If the array is of dimensionality n, then an asterisk may appear in k ≤ n positions. If the jth list position is occupied by an asterisk, the cross section of the array includes elements covered by varying the jth subscript between its bounds. The dimensionality of the cross section is equal to the number of asterisks, k, in the subscript list. If all subscript positions are occupied by asterisks, then this reference to the cross section is equivalent to a reference to the entire array.

The name of the cross section may be used anywhere that the name of an array of dimensionality k is required. Subsequent references to the word "array" in this document should therefore be taken to include cross sections of arrays.

Examples:

1. A (3,*) denotes the third row of the array A.
2. B (*, *, 2) is a two-dimensional cross section and denotes the second plane of the array B.
3. If MATRIX is the array:
   1 2 3
   4 5 6
   7 8 9
   MATRIX (*, 2) is the vector:
   2
   5
   8

ENTRY) may be preceded by a subscripted reference to the label array. Subscripts must be decimal integer constants that may or may not be signed.

The effect of preceding a statement with a subscripted reference is as follows:

1. An INITIAL attribute is constructed for the label array and added to the declaration.
2. A label constant is constructed for the statement carrying the subscripted reference. This label constant is appropriately placed, with respect to the specified subscripts, in the INITIAL attribute.

A label array may not be initialized by using both this form of initialization and the INITIAL attribute in the declaration of the label array in the DECLARE statement.

Example:

```
DECLARE Z (3) LABEL;
    .
    .
    .
Z (1):  IF X > Y THEN GO TO EXIT;
    .
    .
    .
Z (2):  A = A + B + C * D;
    .
    .
    .
Z (3):  A = A + 10;
    .
    .
    .
GO TO Z (I);
```

Statements are given subscripted references (Z (1), Z (2), and Z (3)). Transfer is made to a particular Z by means of giving I the value of 1, 2, or 3.

PL/I DECLARATIONS

Names are described through the use of DECLARE statements. Entire classes of identifiers are described through the use of IMPLICIT statements.

DECLARE STATEMENTS

Function:

The DECLARE statement is a non-executable statement used for the specification of attributes of simple names.

General Format:

```
DECLARE [level] name [attribute] ...
 [, [level] name [attribute] ...] ...;
```

Syntax rules:

1. Any number of identifiers may be declared as names in one DECLARE statement and must be separated by commas. A particular name may be declared only once in a DECLARE statement.
2. Attributes must follow the names to which they refer. (Note that the above description does not show factoring of attributes.)
3. "Level" is a decimal integer used in structure declarations (see "Structures," in Chapter 2). A blank may appear optionally after the level number.

General Rules:

1. All of the attributes for a particular name must be declared together in one DECLARE statement.
2. Attributes of EXTERNAL names, declared in separate blocks and compilations, must not conflict or supply explicit information that was not explicit or implicit in other declarations.

Example:

```
DECLARE JOE FLOAT, JIM FIXED (5,3),
   JACK BIT (10);
```

JOE is declared to be a floating-point scalar variable, JIM a five-position, fixed-point scalar variable with three places to the right of the decimal, and JACK a scalar variable of ten bits.

Factoring of Attributes

Attributes common to several name declarations can be factored. This is achieved by enclosing the subset of name declarations in parentheses and following it by the list of factored attributes, separated by blanks. Factoring may be nested to any desired level. If a factored attribute is in conflict with an attribute specified in the region of factoring, then the inner attribute overrides the factored attribute. Attributes in a list of factored attributes on the same factoring level may not be in conflict with one another. If a factored attribute cannot legally be applied to a name declaration in the region of factoring, then it is not taken to apply to that name declaration.

A level number may be factored before a parenthesized list of declarations. If the name declarations in the region of factoring have level numbers, these override the factored number.

Examples:

The following two DECLARE statements are identical in meaning; the STATIC attribute has been factored in the second DECLARE statement:

```
DECLARE B STATIC, C CHARACTER (5)
   STATIC, D BINARY STATIC, E STATIC;
DECLARE (B, C CHARACTER (5), D BINARY,
   E) STATIC;
```

IMPLICIT STATEMENTS

Function:

Attributes declared in a DECLARE statement are associated with a particular name. Attributes declared in an IMPLICIT statement are associated with a specified set of identifiers. The IMPLICIT statement specifies that all undeclared or partly declared names commencing with a particular letter, which appears in the IMPLICIT statement (as a single letter or as a letter range), have the attributes associated with that letter as specified in the implicit declaration.

A *partly declared* name is one that has been given in a DECLARE statement with insufficient attribute specification for it to be fully defined. In this case, explicitly declared attributes override conflicting implicit ones.

General format:

```
IMPLICIT letter [-letter] attribute...
   [,letter [-letter] attribute...]
   ...;
```

Syntax rules:

1. The format is identical to the DECLARE statement except that letters or letter ranges are used instead of names and that level numbers are not permitted.
2. When a letter range (letter-letter) is given, the first letter must be of lower alphabetic order than the second. (If the 60-character set is used, this would mean alphabetic order in the extended alphabet.)
3. Factoring of attributes is permitted.
4. The following applies if the dimension attribute is used:
   a. It must immediately follow a letter or a letter range.
   b. It may not be factored.
   c. It must specify constant bounds.

General rules:

1. As in the DECLARE statement, if a factored attribute is in conflict with an attribute in the region of factoring, the inner attribute overrides the factored attribute. Attributes on the same level cannot be in conflict with one another.
2. The scope of an implicit statement is an external procedure.
3. No attribute that requires a name known to the procedure is permitted in an IMPLICIT statement. Thus, for example, the attributes LIKE and DEFINED may not be used, because their use requires the appearance of a variable name.

Examples:

1. IMPLICIT C BINARY COMPLEX,
   (B-G EXTERNAL, U-W STATIC, I EXTERNAL,
   J) INITIAL (0);
2. IMPLICIT P-R CHARACTER (2) INITIAL
   (20), S-U (100,100) EXTERNAL;

THE ATTRIBUTES

Attributes are used to give characteristics to their associated identifiers. The attributes of the language are divided into the following classes:

- Data attributes
- Dimension attribute
- SECONDARY attribute
- ABNORMAL/NORMAL attributes
- USES and SETS attributes
- Entry name attributes
- Scope attributes
- Storage Class attributes
- ALIGNED and PACKED attributes
- DEFINED attribute
- INITIAL attribute
- Symbol table attributes
- Structure attributes
- LIKE attribute
- File description attributes

DATA ATTRIBUTES

Arithmetic Data

Variables are declared to be of arithmetic type if they are given any of the attributes base, scale, mode, or numeric picture.

Base

Function:

The base attribute specifies that the data is in binary or decimal form.

General format:

BINARY|DECIMAL

Rules:

This attribute may not be specified in combination with the PICTURE attribute.

Default:

See "Default Conditions for **Arithmetic** Data."

Examples:

DECLARE A DECIMAL, B BINARY;

Scale

Function:

The scale attribute specifies that the data is in fixed-point or floating-point form.

General format:

FIXED|FLOAT

Rules:

This attribute may not be given in combination with the PICTURE attribute.

Default:

See "Default Conditions for Arithmetic Data."

Examples:

```
DECLARE A FIXED, B FLOAT;
```

Mode

Function:

The mode attribute specifies that the mode of the data is real or complex.

General format:

REAL|COMPLEX

Rules:

This attribute may be given in combination with the PICTURE attribute, to specify a complex numeric field.

Default:

See "Default Conditions for Arithmetic Data."

Example:

```
DECLARE A COMPLEX, B REAL:
```

Precision

Function:

The precision attribute specifies the number of significant binary or decimal digits to be maintained for both fixed-point and floating-point data, as well as the scale of the data.

General format:

(number-of-digits [,scale-factor])

Rules:

1. The precision attribute must immediately follow a scale, base, or mode attribute and may never appear alone or separated from one of these attributes.
2. "Number-of-digits" is the number of binary or decimal digits to be maintained and is used with both fixed-point and floating-point data.
3. The "scale-factor" defines the position of the point with respect to an integer data item of the specified number of digits. It is used only with fixed-point data.
4. Both of the above described fields must be decimal integer constants.
5. When the scale is fixed and no scale factor is given, it is assumed to be zero.
6. The scale factor may be negative, and it may be larger than the number of digits.
7. The scale factor effectively multiplies the integer data by the base raised to the power of the scale factor with the sign reversed. For example, decimal data of precision (5,2) represents numbers from .01 to 999.99 in magnitude; decimal data of precision (5,-2) represents numbers from 100 to 9999900 in magnitude.

Default:

Default is defined separately for each particular implementation of the language.

Examples:

```
DECLARE A FLOAT (3), B REAL (10)
   FLOAT, X FIXED (5,2);
```

The following table shows the meaning of the scaling for fixed-point variables:

Integer	Scale	Precision	Value
1. 00123	FIXED	(5,2)	1.23
2. 00123	FIXED	(5,-2)	12300
3. 123	FIXED	(3,4)	.0123
4. 123	FIXED	(3,-4)	1230000

Default Conditions for Arithmetic Data

If the base, scale, and mode are not specified, the arithmetic default attributes are dependent upon the first letter of the name. If the first letter of the name is I through N, FIXED REAL BINARY is assumed; otherwise, FLOAT REAL DECIMAL is assumed.

If arithmetic data attributes are partly specified, the omitted attributes are assumed according to the following table:

Specified	Assumed		
	Base	Scale	Mode
BINARY		FLOAT	REAL
DECIMAL		FLOAT	REAL
FIXED	DECIMAL		REAL
FLOAT	DECIMAL		REAL
REAL	DECIMAL	FLOAT	
COMPLEX	DECIMAL	FLOAT	
BINARY FIXED			REAL

The table implies the other combinations that may be specified. Note the BINARY FIXED combination that is included.

If precision is not specified, the assumed precision is that which is defined for the particular implementation of the language that is being used.

The PICTURE Attribute

Function:

The PICTURE attribute is used to define the internal and external formats of numeric and character-string data field and to specify the editing of data. This discussion is limited to the use of the PICTURE attribute with numeric data. The use of the PICTURE attribute with character-string data is described in "String Attributes." The picture characters are described in Appendix 2.

General format:

PICTURE 'numeric-picture-specifications'

General rules:

1. PICTURE may not be specified in combination with the base and scale attributes.

   Numeric fields have mode, base, scale, and precision; these are specified by the picture characters used in describing the field, and by the use of the mode attribute if COMPLEX. Note the exception that sterling pictures are treated as a separate category, although they are real fixed-point decimal fields.
2. A "picture specification" is composed of a string of picture characters. It must be enclosed in quotation marks. Individual picture characters may be preceded by an iteration factor, which is an integer n, enclosed in parentheses, to indicate repetition of the character n times.
3. The following paragraphs indicate the combination of picture characters that show mode, scale, base, and precision. In this discussion, a fixed-point field has one field, and a floating-point field has two subfields. Only one sign character can appear in a subfield.
   a. Real binary fixed-point fields may contain picture characters S, 1, 2, 3, and V. They take the general forms:

      PICTURE '[S] 1... [V] 1...[F([+|-] integer)]'
      PICTURE '2...[V] 2...[F([+|-] integer)]'
      PICTURE '3...[V] 3...[F([+|-] integer)]'

      Only one V, representing a point, may be present in a picture specification, but it may be in any position. When a sign character (S) is specified, the field will contain a binary 1 if the value is negative or a 0 if the value is zero or positive. Only one sign character is permitted in a subfield.
   b. Real binary floating-point fields may contain picture characters S, 1, 2, 3, V, and K. They take the general forms:

      PICTURE '[S] 1... [V] 1... K[S] 1...'
      PICTURE '2... [V] 2... K2...'
      PICTURE '3... [V] 3... K3...'

      The second sign character allowed in the first form represents the sign of the exponent.
   c. Real decimal fixed-point fields may contain all picture characters except A, X, E, K, 1, 2, 3. With the exception of sterling fields (see item f, below) and characters allowed for them, they take the general form:

      PICTURE '9... [V] 9...[F[+|-] integer]'

      Sign, editing, and zero-suppression picture characters, as explained in Appendix 2, may be included. The V may appear only once in a picture specification. If no V is given, the decimal point will be assumed to appear to the right of the last digit. No attempt has been made to show the use of all valid picture characters in the general format above. These are explained in Appendix 2.

d. Real decimal floating-point fields may contain all picture characters except A, X, 1, 2, 3, and sterling picture characters.

They take the general form:

```
PICTURE '9...  [V] 9...{E|K}
   9...'
```

Sign, editing, and zero-suppression picture characters may be included. Sign characters refer to the subfield in which they appear, except a CR or a DB, which refers to the first subfield.

e. Complex fields may contain those picture characters that are valid for real fields as described above. They take the general form:

```
real-picture
```

The "real-picture" represents both portions of the complex number. The real-picture may not specify a sterling field.

f. Sterling fields are considered to be real fixed-point decimal fields. When involved in arithmetic operations, they will be converted to a value representing fixed-point pence. A sterling picture field may contain all picture characters except 1, 2, 3, K, E, X, and A. Sterling pictures have the general form:

```
PICTURE 'G editing-character-1
   pounds-field separator-1
   shillings-field separator-2
   pence-field-1 [V|V.|.V
   [pence-field-2]]
   [editing-characters-2]'
```

"Editing character 1" may be one or more of the following picture characters:

```
$ + - S
```

The "pounds field" may contain the following picture characters:

```
Z Y * 9 T I R $ + - S
```

"Separator 1" may be one or more of the following picture characters:

```
/ .  B V
```

The "shillings field" may be:

```
{99|ZZ|Y9|Z9|ZY|8}
```

The 9s may be replaced by T, I, or R.

The picture character Z may occur only if the whole of the field to the left of this character (including the pounds field) is also suppressed using the character Z.

"Separator 2" may be one or more of the picture characters:

```
/ .  B V H
```

The "pence field" takes the form:

```
{99|ZZ|Y9|7|Z9|ZY|6}  [V|V.
[9|Z|Y ...]]  [D]  [CR|DB|S|-|+]
```

Any of the nines may be replaced by one of the following:

```
T I R
```

In a sterling picture, there can be only one of the following characters:

```
T I R CR DB S + -
```

Zero suppression characters can only appear after the decimal point if all digits are suppressed in the field.

4. The precision of picture specifications is described below. In this discussion, the following picture characters, actual and conditional, are defined as digit positions:

```
1 2 3 9 Z * Y T I R
  and the drifting
        $ S + -
```

The precision of a real fixed-point numeric field is (m,n), where $\underline{m}$ is the total number of digit positions in the field and $\underline{n}$ is the number of digit positions following the V.

The precision of a real floating-point field is (p), where $\underline{p}$ is the total number of digit positions before the E or K.

The precision of a complex field is the precision of one of its two parts.

Decimal or binary fixed-point pictures may have a scaling factor. This may be achieved by placing the following at the extreme right of the picture subfield:

```
F ([+|-] integer)
```

with the "integer" value represented

by $\underline{q}$, this specifies that the decimal or binary point should be assumed to be $\underline{q}$ places to the right (or left, if negative) of the position assumed in the absence of the scaling factor. The precision of the numeric field is then (m,n-g).

These precisions may not exceed the limits for decimal fixed-point values, as defined for the particular implementation of PL/I.

String Attributes

Function:

The string attributes specify string data to be either in bit-string form or in character-string form with a specified length. The form of character-string data may also be specified.

General format:

```
  ⎧ ⎧BIT      ⎫                           ⎫
  ⎪ ⎩CHARACTER⎭  (length)   [VARYING]     ⎪
  ⎨                                       ⎬
  ⎩  PICTURE 'character-picture-          ⎭
                 specifications'
```

Rules:

1. BIT specifies bit-string data, CHARACTER specifies character-string data, and PICTURE specifies character-string data in picture form.
2. The "length" specifies the actual length of fixed-length strings and the maximum length of varying-length strings, in which case the word VARYING is used.
3. The length specification may be an expression or an asterisk.
4. If the length specification is an expression, it will be converted to an integer at the point of allocation or upon entry to the declaring block for parameters.
5. An asterisk may be used when the length is to be taken from a previous allocation for parameters or controlled variables or if it is to be specified in a subsequent ALLOCATE statement for CONTROLLED variables.
6. The length of strings declared STATIC must be a decimal integer constant.
7. Since PICTURE is an attribute that also may apply to arithmetic data, a separate explanation is in the section entitled "The PICTURE Attribute." Additional picture characters are provided when the PICTURE attribute is used to declare character-string data. These may be found in Appendix 2.

Default:

If none of these attributes is specified, coded arithmetic data is assumed.

Example:

```
DECLARE A BIT (10), B CHARACTER (5), C
   PICTURE 'XAA.AA', D BIT(*);
```

A is a field of ten bits; B is a field of five characters; C is a field of characters, letters, and a decimal point; and D is a field of bits with a length to be taken from a previous allocation or is to be subsequently allocated.

The LABEL Attribute

Function:

The LABEL attribute specifies that the associated variable will have statement labels as values. To aid optimization of the object program, it may also specify the values a label variable may have during execution of the program.

General format:

```
LABEL [(statement-label-constant
   [, statement-label-constant]...)]
```

Rules:

1. If no statement-label constants are specified following the LABEL attribute, the value of the variable may be any of the statement labels known in the procedure in which the variable is declared.
2. If the variable is a parameter, the value can be any statement label that can be specified as an argument.
3. If a list of statement-label constants is specified, the variable may have as values only members of the list.

Example:

```
DECLARE START LABEL (LABEL1, LABEL2,
   LABEL3);
```

THE DIMENSION ATTRIBUTE

Function:

The dimension attribute defines the bounds of an array.

General format:

```
(bound [, bound] ...)
```

where "bound" is
{[lower-bound :] upper-bound}|*

Rules:

1. The number of bounds or sets of bounds specify the number of dimensions in an array.
2. Bounds that are expressions are evaluated and converted to integer data when storage is allocated for the array or when linkage is established for parameters.
3. The bounds are indicated as follows:
   a. If only the upper bound is given, the lower bound is assumed to be one.
   b. When the actual bounds for each dimension are to be taken from a previous allocation or are to be specified in a subsequent ALLOCATE statement, an asterisk may be used to represent all of the dimension bounds. Thus, asterisks may be used only for parameters and CONTROLLED variables.
   c. The lower bound must be less than or equal to the upper bound.
4. The bounds of arrays declared static must be decimal integer constants.

Examples:

1. DECLARE TABLEA(5,8), TABLEB(-5:5,10);

   TABLEA is a two-dimensional array with 5 rows and 8 columns (subscripts 1 to 5 and 1 to 8). TABLEB is a two-dimensional array with 11 rows and 10 columns (subscripts -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 for the rows and 1 through 10 for the columns).

2. DECLARE MATRIX (*,*);

   MATRIX is a two-dimensional array. The bounds are to be taken from a previous allocation or are to be subsequently allocated.

THE SECONDARY ATTRIBUTE

Function:

The SECONDARY attribute is used to specify that certain data normally does not require efficient storage.

General format:

SECONDARY

Rules:

1. This attribute may be declared only for major structures, arrays, and variables not contained in structures or arrays.
2. The attribute specifies that where possible and necessary, less than normally efficient storage may be allocated to the variable.

THE ABNORMAL AND NORMAL ATTRIBUTES

Function:

The ABNORMAL and NORMAL attributes are used to specify procedures and/or data as being either normal or abnormal.

General format:

ABNORMAL|NORMAL

Rules for abnormality of procedures:

1. Abnormality is a property of both external and internal procedures. Blocks invoking procedures that are abnormal must declare those names with the ABNORMAL, USES, or SETS ATTRIBUTES. These attributes enable program optimization to be performed.
2. An <u>external</u> procedure is abnormal if it, or any procedures invoked by it:
   a. Access, modify, allocate or free external data.
   b. Modify, allocate, or free their arguments.
   c. Return inconsistent function values for the same argument values.
   d. Maintain any kind of history.
   e. Perform I/O operations.
   f. Return control from the procedure by means of a GO TO statement.
3. An <u>internal</u> procedure is abnormal:
   a. Under any of the conditions listed above for external procedures.
   b. If it, or any procedures called by it, access, modify, allocate, or free variables declared in an outer block.
4. Abnormal external procedures invoked as functions must be declared in the invoking block with at least one of the attributes, ABNORMAL, USES, or SETS.
5. ABNORMAL used alone specifies that all possible types of abnormality should be assumed. It is unnecessary to specify ABNORMAL for the built-in functions, TIME and DATE.
6. The NORMAL attribute specifies that the entry name is for a procedure that is not abnormal. This attribute is

used to override a factored or implicit ABNORMAL attribute.

Rules for abnormal data:

1. The ABNORMAL attribute may be declared for any variable.
2. The ABNORMAL attribute specifies that a variable may be altered or otherwise accessed at an unpredictable time during the execution of a program. This situation might occur, for example, during the execution of an ON-unit as described in "The ON Statement," in Chapter 6.
3. Every time ABNORMAL data is referred to, its associated storage contains its current value.
4. The NORMAL attribute is used to override a factored or implicit ABNORMAL attribute.

Default for abnormality of procedures:

If an <u>external</u> entry name appears as a function reference, the entry name is assumed to have the NORMAL attribute; otherwise, the attribute is assumed to be ABNORMAL. Entry names of all internal procedures and entry names of external procedures invoked in CALL statements are assumed to have the ABNORMAL attribute.

Default for abnormal data:

Variables are assumed to be NORMAL.

THE USES AND SETS ATTRIBUTES

Function:

The USES and SETS attributes are used to specify, for an entry name, the nature of an abnormality due to data manipulation.

General format:

```
USES (item[,item]...)
SETS (item[,item]...)
```

Rules:

1. The USES and SETS attributes may be declared in an invoking block for any entry name used to invoke a procedure. If the ENTRY attribute is not declared, ENTRY is implied. If either USES or SETS is declared in the invoking procedure, complete information must be given about the data that is used and/or set by the invoked procedure.
2. The items of the list following a USES or SETS attribute may be as follows:
   a. A decimal integer <u>n</u>, specifying the nth argument of any invocation of the procedure at the named entry point.
   b. An external, unsubscripted data name known to both the block containing the declaration and the invoked procedure.
   c. An internal, unsubscripted data name known to both the block containing the declaration and the invoked procedure.
   d. An asterisk indicating <u>all</u> identifiers described in b and c, above.
3. An item in the USES list specifies the following:
   a. That the invoked procedure or procedures invoked by it access that item.
   b. That neither the invoked procedure nor procedures invoked by it reassign that item unless it is also specified in a SETS attribute.
   c. That neither the invoked procedure nor procedures invoked by it access any <u>other</u> data known to the block, except data designated by explicit arguments in either a CALL statement, a statement with a CALL option, or a function reference.
4. An item in the SETS list specifies the following:
   a. That the invoked procedure or procedures invoked by it reassign, allocate, or free that item.
   b. That neither the invoked procedure nor procedures invoked by it access that item other than to reassign, allocate, or free it, unless it is also specified in a USES attribute.
   c. That neither the invoked procedure nor procedures invoked by it reassign, allocate, or free any <u>other</u> data known for the block, except data designated by explicit arguments in the case of a CALL statement.
5. If an item in a USES or SETS list, as described in 2b and 2c, above, is defined on a base (see "The DEFINED Attribute") and if the base and any other items defined on it are known both to the invoking and invoked blocks, the base and the other items must also be specified in the list.
6. A structure name appearing in a USES and SETS list implies that the names of all items contained in the structure also are on the list. It does not imply that items defined on elements of the structure are in the list; these must be declared as in rule 5, above.
7. The USES attribute specifies that the procedure is abnormal for those variables in the USES list.

8. The SETS attribute specifies that the procedure is abnormal for those variables in the SETS list.
9. If the USES and SETS attributes are given and the invoked procedure is abnormal in any other way, the ABNORMAL attribute must still be given (unless default). If the USES and SETS attributes are given and the invoked procedure is not otherwise abnormal, the ABNORMAL attribute should not be given.

ENTRY NAME ATTRIBUTES

An entry name is declared by giving it the ENTRY attribute. A name may be declared to have any of the attributes SETS, USES, GENERIC, and BUILTIN. These attributes all imply ENTRY and thus ENTRY need not be specified. The entry name also may have the attributes ABNORMAL or NORMAL and, with the exception of LABEL, any of the data attributes listed in the beginning of this section.

The data attributes specify the characteristics of the value returned when the entry name is invoked as a function. If data attributes are not specified, default or implicit characteristics will be assumed (see "Relationships of Declared, Implicit, and Default Attributes," in this chapter).

The ENTRY Attribute

Function:

The ENTRY attribute is used to declare, within a procedure, entry names that are referred to in that procedure.

General format:

```
ENTRY[ (parameter-attribute-list
   [,parameter-attribute-list] ...)]
```

Rules:

1. When ENTRY is used without a "parameter attribute list," it specifies that the identifier being defined is an entry name. An entry name must be declared with the ENTRY attribute unless a reference is made to the entry name in a CALL statement or in a function reference with arguments or if it is declared to have any of the attributes SETS, USES, GENERIC, and BUILTIN. ENTRY without a parameter attribute list specifies nothing about the number or nature of the parameters.
2. When ENTRY is used with parameter attribute lists, each parameter attribute list is a succession of attributes describing the parameters of the entry point. Permitted attributes are those allowed for parameters.
3. The number of parameter attribute lists must be the same as the number of parameters required by the entry point. If a parameter attribute list is null, its place must be kept by a comma.
4. Parameter attribute lists are not necessary if the parameters of the entry name are not to be described.
5. The dimension attribute may be specified for array parameters. It must be the first attribute specified for the parameter.
6. The structuring for a structure parameter is specified by a structure description using level numbers without identifiers, the level number being immediately followed by the list of attributes for that level of the structure. The first item in the description of the structure parameter must be at level one.
7. Expressions occurring in ENTRY attributes for length or dimension bounds are evaluated upon entering the block to which the declaration of the ENTRY attribute is internal. If an argument position specifies an entry with no data attributes, no default data attributes are provided.

Default:

If no attributes are given for a parameter, no assumptions are made about it. When attributes are required for a full definition of a parameter but are not specified, the required attributes are deduced according to the default rules given in "Relationships of Declared, Implicit, and Default Attributes."

The GENERIC Attribute

Function:

The GENERIC attribute is used to define the entry name as a family of entry names, each of which is referred to by the entry name being declared. When the generic name is referred to, the proper entry name is selected, based upon the arguments specified for the generic name in the procedure reference.

General format:

```
GENERIC (entry-name-declaration
   [,entry-name-declaration] ...)
```

Rules:

1. No other attributes may be specified for the name being given the GENERIC attribute.
2. Each "entry name declaration" following the GENERIC attribute corresponds to one member of the family.
3. Each entry name declaration must have an entry name attribute. It may optionally have ABNORMAL and/or data attributes. No entry name declaration may have the GENERIC attribute.
4. Each entry name declaration must specify attributes for every parameter of the associated entry name. Attributes unspecified but required for full definition will be deduced from default rules.
5. When a generic name is referred to, the attributes of the arguments must match exactly the list following the entry name declaration of one <u>and only one</u> member of the family. The reference is then interpreted as a reference to that member. Thus, the selection of a particular entry name is based upon the arguments of the reference to the generic name.
6. When arrays are involved, the selection is based only upon the dimensionality of the array, not upon extents. When strings are involved, the lengths do not participate.

Example:

```
DECLARE
  CALCULATE GENERIC (FIXCALC ENTRY (FIXED),
  FLTCALC ENTRY (FLOAT)), Y FLOAT
  INITIAL (50);
X=Y + CALCULATE (Y);
```

The assignment statement results in the invocation of the procedure FLTCALC, since the argument Y matches the entry attribute of the FLTCALC member of the family.

<u>The BUILTIN Attribute</u>

Function:

The BUILTIN attribute specifies that the reference to the associated identifier within the scope of the declaration is interpreted as a reference to the built-in function of the same name. It is used only when necessary to avoid ambiguity concerning the name of the built-in function.

General format:

```
BUILTIN
```

Rules:

1. BUILTIN is used to refer to a built-in function in a block that is contained in another block in which the built-in function name has been declared to have another use.
2. If the BUILTIN attribute is declared for an entry name, it may have no other attributes.
3. For a list of built-in functions see Appendix 1.

SCOPE ATTRIBUTES

Function:

The scope attributes are used to specify the scopes in which declared identifiers are known.

General format:

$$\begin{Bmatrix} \text{INTERNAL} \\ \text{EXTERNAL [(identifier)]} \end{Bmatrix}$$

Rules:

1. For a full discussion of the INTERNAL and EXTERNAL attributes, see "Scope of Declarations," in Chapter 1.
2. In the form EXTERNAL (identifier), the identifier specifies a heading for the declared name. Subsequently, when references are made to the declared name, they are limited to the union of scopes of the same name with the same heading.

Default:

If the scope is unspecified for variable names, INTERNAL is assumed.

Example:

```
DECLARE SUM EXTERNAL (X);
```

The variable SUM is declared external with the heading X. In subsequent references, the heading distinguishes this variable from other variables named SUM with no heading or other headings.

STORAGE CLASS ATTRIBUTES

Function:

Storage class attributes are used to allocate a particular class of storage to variables.

General format:

STATIC|AUTOMATIC|CONTROLLED

Rules:

1. STATIC specifies that storage is allocated at the start of execution of the program and is not released until program execution has been completed.
2. AUTOMATIC specifies that storage is allocated on each entry to the block to which its declaration is internal. The storage is released on leaving the block. If the block is a procedure that is invoked recursively, the previously allocated storage is pushed down on entry, and the latest allocation of storage is popped up on return.
3. CONTROLLED specifies that full control will be maintained over the allocation and freeing of storage by means of the statements ALLOCATE and FREE.
4. AUTOMATIC variables may have internal scope only. STATIC and CONTROLLED variables may have internal or external scope.
5. Storage class attributes may not be specified for entry names and file names.
6. STATIC and AUTOMATIC attributes may not be specified for parameters.
7. STATIC and CONTROLLED attributes may not be specified for label variables.
8. Variables declared with adjustable lengths and dimensions may not have the STATIC attribute. However, STATIC arguments can be passed as parameters with adjustable lengths or dimensions.
9. If a procedure involving static storage is invoked from within or as a separate task, the static storage is common to all invocations.
10. If, during execution of a statement, controlled data is allocated or freed (by an abnormal function, for example), any reference in the statement to that data produces an undefined result.
11. All items in a structure must be of the same storage class. If a major structure is given a storage class attribute, this applies to all elements of the structure. If a structure is CONTROLLED, only the major structure, and not the elements, may be allocated and freed.

Default:

1. If storage class is unspecified and the scope is EXTERNAL, STATIC is assumed.
2. If storage class is unspecified and the scope is INTERNAL, AUTOMATIC is assumed.
3. If neither storage class nor scope is specified, AUTOMATIC is assumed.

Example:

```
EXAMPLE: PROCEDURE;
         DECLARE A STATIC INITIAL
           (0), B CONTROLLED, C(10);
         ALLOCATE B;
         A = A + 1;
             .
             .
             .
         FREE B;
         END EXAMPLE;
```

The variable A is of the static storage class and is used to count the number of times the procedure is invoked. The variable B is of the controlled storage class, and storage is allocated and freed by use of the ALLOCATE and FREE statements. The variable C is of the automatic storage class by default.

THE ALIGNED AND PACKED ATTRIBUTES

Function:

The ALIGNED and PACKED attributes are used to specify in storage the arrangement of string data elements within data aggregates.

General format:

ALIGNED|PACKED

Rules:

1. These attributes may be *specified* for the following:
   a. Names of major structures.
   b. Names of arrays that are not themselves part of a structure.
2. PACKED specifies that each string element is packed in storage contiguous with the string elements that surround it. There should be no unused storage between two adjacent string elements.
3. ALIGNED specifies that each string data element within the aggregate may start at a storage boundary to be defined individually for each implementation of PL/I. This implies that two adjacent string elements may not occupy contiguous storage.

Default:

1. The default for major structures is PACKED.
2. The default for arrays that are not part of structures is ALIGNED.

Examples:

```
DECLARE
    1 A (10) BIT (1000) PACKED, 2B BIT
    (200), 2 C BIT (500), 2 D BIT
    300), E (10,15) ALIGNED BIT (15);
```

The elements of the major structure A will occupy a continuous area of storage. Each element of the array E will start at a storage boundary defined for that implementation of PL/I. There may be unused storage between the elements of the array.

THE DEFINED ATTRIBUTE

Function:

The DEFINED attribute specifies that scalar, array, or structure data is to occupy the same storage as that already assigned to other data.

General format:

```
defined-item DEFINED base-identifier
    [POSITION decimal-integer-constant]
```

Rules for defining:

1. In general, the "defined item" must have the same characteristics as the "base identifier." However, mixed defining is permitted in the following two classes:
   a. The bit class is composed of the following:
      (1) numeric fields of binary base
      (2) fixed-length bit strings
      (3) packed arrays of either (1) or (2)
      (4) packed structures of either or both (1) and (2)
   b. The character class is composed of the following:
      (1) numeric fields of decimal base
      (2) fixed-length character strings
      (3) packed arrays of either (1) or (2)
      (4) packed structures of either or both (1) and (2)
2. The following attributes are inherited by the defined item from the base identifier and need not be specified for the item having the DEFINED attribute:
   a. data attributes, except for the length of strings or PICTURE (see the rules for mixed defining, listed in this section).
   b. the storage-class attributes STATIC, CONTROLLED, AUTOMATIC
   c. SECONDARY
   d. ABNORMAL, NORMAL
   e. ALIGNED, PACKED

   If these are specified for the defined item, they must match those of the base identifier.
3. The length of strings and the PICTURE attribute will be inherited by the defined item from the base identifier if they are not specified.
4. The INITIAL, SYMBOL, and scope attributes must not be specified for the defined item. It should be noted that although the base may have the EXTERNAL attribute, the defined item cannot. The name of the base, if declared external, will be known in all blocks in which it is declared external, but the name of the defined item will not. However, the value of the defined item will be changed if the value of the base item is changed in an external block.
5. The LIKE attribute must be specified for a defined item if it is to apply.
6. The defined item must always be specified as a subset (including the full set) of the base identifiers.
7. Expressions specified in base-identifier subscript lists are evaluated when the defined item is referred to and not when it is declared. Use of a defined item in an argument list is interpreted as a reference to the defined item.
8. The base identifier must always be known within the block where the defined identifier is declared and cannot have been declared with the DEFINED attribute.
9. If the attributes of the defined data involve expressions (other than in a defining subscript list, as discussed in the rules for subscripted array defining), these expressions are evaluated on entry to the declaring block, regardless of the storage class of the base identifier. The base identifier always is taken as the current generation at each reference to the defined variable.

Rules for scalar defining:

1. Both the defined item and base identifier must be scalars.
2. The base identifier may be subscripted, in order to specify a scalar element of an array, but the defined term may not be an element of a structure or an array.
3. Permitted forms are as follows:

Defined Item	Base Identifier
coded arithmetic	coded arithmetic of the same base, scale, mode, and precision
label	label

binary numeric field or bit string	binary numeric field or bit string
decimal numeric field or character string	decimal numeric field or character string

4. The POSITION attribute may be specified only when the base is a fixed-length string. It specifies the position (n) relative to the start of the base where the defined item begins. If omitted, Position (1) is assumed. The position (n) is restricted as follows: n + length (defined-item) - 1 must be less than or equal to the length (base identifier).

Rules for array defining:

1. Both the defined item and the base identifier must be arrays.
2. The defined item must have a dimension attribute, and may not be an element of a structure.
3. The permitted forms are the same as those for scalar defining.
4. In array defining there is a relationship between each element of the defined array and a corresponding element of the base identifier.
5. The elements of the defined array must have lengths less than, or equal to, the lengths of the base array.
6. The POSITION option may be given when the base identifier refers to an array of strings. It specifies that each element of the defined array begins at the nth bit or character position of the corresponding element of the base array.
7. Two classes of array defining are permitted, simple and subscripted.

Rules for <u>simple</u> array defining:

1. The base identifier must be an unsubscripted array name having the same number of dimensions as the defined array.
2. The dimension bounds of the defined array must be a subset of the bounds of the base array.
3. A subsequent subscripted reference to the defined array is interpreted as a reference to the base array with identical subscripts.
4. A subsequent unsubscripted reference to the defined array is interpreted as a reference to the declared subset of the base array specified by the dimension bounds.

Rules for <u>subscripted</u> array defining:

1. The base identifier must be an unsubscripted array name followed by a defining subscript list. The base array need not have the same number of dimensions as the defined array.
2. The defining subscript list defines the relationship between the elements of the defined array and the base array and must have as many subscripts as the base array has dimensions.
3. The defining subscripts may be an expression, including the dummy variable iSUB, and take the following form:

   $a \pm a_1$ *1SUB$\pm$ a_2 *2SUB... $\pm$ a_n *nSUB

   In the expression iSUB, <u>i</u> is a decimal integer constant in the range 1 to <u>n</u>, with n being the dimensionality of the array. The symbol <u>a</u> is any scalar expression involving variables known within the block containing the DEFINED declaration. The integer value of the expression will be used. If any <u>a</u> is zero, that iSUB may be omitted.
4. The subscripted reference to the defined array is then interpreted as follows:
   a. Each iSUB in the defining subscript is replaced by the integer value of the ith subscript given for the defined array. Before replacement, the subscript conceptually is enclosed in parentheses.
   b. The reference to the defined array elements is a reference to the base array element specified by the generated subscript.
5. For example, in the following statement, B is a vector having as elements the diagonal of matrix A:

```
DECLARE    A(10,10),   B(10)   DEFINED
   A(1SUB, 1SUB);
```

6. A subsequent unsubscripted reference to the defined array is interpreted as a reference to the entire array as defined by the mapping.
7. If a defined array name is specified as an argument to an invoked procedure, the expressions in the defining subscript list are evaluated before the invocation. The invoked procedure can still reassign values to elements of the defined array by a parameter, but the relationship between the defined array elements and the base elements is frozen on entry.
8. The POSITION option cannot be used with subscripted array defining.

Rules for mixed defining (structures and arrays):

1. Major structures and arrays not contained in structures, having elements all of the same class as described

earlier in this section, may be defined on scalar strings of the same class and on structures having elements all of the same class and which are not part of a major structure having the ALIGNED attribute.

2. When scalar strings are defined items, base items may be any of the following:
   a. major structures that are packed
   b. minor structures not contained in dimensioned structures but contained in a packed major structure
   c. structure elements with the base being specified as a subscripted structure name
   d. unsubscripted packed arrays not contained in dimensioned structures
3. All of the elements of the base item must be of the same class as the defined string.
4. When the base item is a scalar string, the POSITION option may be specified to indicate that the defined array or structure is offset from the start of the string. It may not be specified with mixed defining when the base is an array or structure.
5. Defining subscript lists may not be used with mixed defining.

Examples:

```
DECLARE 1P, 2Q CHARACTER (10), 2R
     CHARACTER (100),
   PSTRING1 CHARACTER (110) DEFINED P;

DECLARE LIST CHARACTER (40), ALIST
     CHARACTER (10)
   DEFINED LIST, BLIST CHARACTER (20)
     DEFINED LIST
   POSITION (21), CLIST CHARACTER (10)
     DEFINED LIST
   POSITION (11);

DECLARE ALL (16), EVEN (8) DEFINED ALL
      (2*1SUB);
```

THE INITIAL ATTRIBUTE

Function:

The INITIAL attribute either specifies constant values to be assigned to data when storage is allocated to it, or it specifies a procedure to be invoked to perform initialization at allocation.

General format:

1. INITIAL (item[, item] ...)
2. INITIAL CALL entry-name [argument-list]

Rules for form 1:

1. In this discussion, the term constant denotes either a constant or a complex expression of the following form:

   real-constant {+|-} imaginary-constant

2. One constant value is required for a scalar; more may be given for an array.
3. Constant values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly).
4. If too many constant values are specified, excess ones are ignored; if not enough are specified, the remainder of the array is not initialized.
5. The items in the list may be an optionally signed constant, an asterisk denoting no initialization for a particular element, or an iteration specification.
6. The iteration specification has one of the following general forms:

(iteration-factor) constant

(iteration-factor) (item {, item} ...)

(iteration-factor) *

7. The "iteration factor" may be any expression that satisfies the rules stated in the section on "Prologues" in Chapter 8. When storage is allocated, the expression is evaluated to give an integer that specifies the number of repetitions.
8. Only constants are permissible as iteration factors for STATIC data.
9. Iterations may be nested.
10. See "Statement Label Data," in Chapter 2, for an alternative method of specifying initial values for label arrays.
11. The INITIAL attribute may not be given for the following:

    entry names
    file names
    DEFINED data
    structures
    parameters

Rules for form 2:

1. The entry name and arguments passed must satisfy the conditions stated in "Prologues."
2. This form may not be used to initialize static data.

General rules:

1. Initialization of data must be done within the procedure that will allocate storage for the data. For exam-

ple, CONTROLLED data initial values must be specified within each procedure that will allocate the data.

Examples:

```
DECLARE SWITCH INITIAL ('1' B);

DECLARE MAXVALUE INITIAL (99), MINVA-
   LUE INITIAL (-99);

DECLARE A (100, 10) INITIAL ((920) 0,
    (20) ((3) 5, 9));

DECLARE TABLE (20,20) INITIAL CALL
   INITIALIZE (X, Y);
```

In the last example INITIALIZE is the name of a procedure that sets the initial values of the 20 by 20 array TABLE. X and Y are arguments that are passed on to INITIALIZE.

SYMBOL TABLE ATTRIBUTES

Function:

The symbol table attributes are used in conjunction with data-directed input/output operations. They specify whether or not the names of data-directed input/output elements are to be placed in a symbol table.

General format:

$$\left\{\begin{array}{l}\text{SYMBOL [(identifier)]} \\ \text{NOSYMBOL}\end{array}\right\}$$

Rules:

1. SYMBOL specifies that the declared name is to be placed in the symbol table.
2. SYMBOL (identifier) is used when the declared name must be qualified to make it unique. This specifies that the identifier in parentheses is to appear in the symbol table as a synonym for the name to which it refers.
3. A variable whose name or synonym appears in the symbol table may have its values transmitted under data-directed output.
4. NOSYMBOL specifies that the declared name is not to appear in the symbol table.
5. A name will not appear in the symbol table unless it is given a SYMBOL attribute or, alternatively, unless it appears in a list for data-directed input or output.

Reference:

See Chapter 5 for a complete discussion of data-directed input and output.

Default:

The default attribute is NOSYMBOL unless the name appears in a list for data-directed input or output.

THE LIKE ATTRIBUTE

Function:

The LIKE attribute specifies that the name being declared is given the same structure as the name following the attribute LIKE.

General format:

```
LIKE structure-name
```

Rules:

1. The "structure name" may be unqualified or qualified, but it may not be subscripted.
2. The structure must be known to the block containing the LIKE attribute.
3. Neither the structure name nor any of its substructures can be declared with the LIKE attribute.
4. The LIKE attribute specifies that the name being declared is a structure with a substructure having elements with attributes and names identical to the names and attributes of the elements of the named structure.
5. If the structure description of the named structure has been declared, and if a direct application of the description to the structure being declared LIKE would cause an incorrect discontinuity in level numbers, then the level numbers will be modified by a constant subtrahend before application.
6. No attributes of the structure other than the substructure are transferred to the name being declared.
7. The member that immediately follows the member that has the LIKE attribute must have a level-number that is equal to or less than that of the member that has the LIKE attribute.

Examples:

```
1.  DECLARE 1 A,
            2 FIELD1,
              3 DTL1 PICTURE '$ZZ.99',
              3 DTL2 CHARACTER (10),
```

```
         2 FIELD2 BIT (50),
        1 X,
         2 FIELD1,
           3 SUBFLD1 LIKE A . FIELD1,
           3 TABLES (3),
         2 FIELD2 LIKE A . FIELD1;
```

The above is equivalent to:

```
DECLARE 1 A,
         2 FIELD1,
         3 DTL1 PICTURE '$ZZ.99',
           3 DTL2 CHARACTER (10),
          2 FIELD2 BIT (50),
        1 X,
         2 FIELD1,
           3 SUBFLD1,
            4 DTL1 PICTURE '$ZZ.99',
             4 DTL2 CHARACTER (10),
           3 TABLE (3),
         2 FIELD2,
           3 DTL1 PICTURE '$ZZ.99',
           3 DTL2 CHARACTER (10);
```

2. `DECLARE 1 A EXTERNAL, 2(B,C,D), 1 E LIKE A;`

The above is equivalent to :

```
DECLARE 1 A EXTERNAL, 2(B,C,D), 1 E,
      2(B,C,D);
```

FILE DESCRIPTION ATTRIBUTES

File description attributes are used to describe data files. Declaration of the same file in more than one external procedure must not conflict. For a complete discussion of data files see Chapter 5.

The FILE Attribute

Function:

The FILE attribute specifies that the associated identifier is a file name.

General form:

FILE

Standard Attributes

Function:

The standard attributes are used to assign a file name as a synonym for the standard input/output file.

General format:

STANDIN|STANDOUT

Rules:

No other attributes except the ZERO attribute may be used to describe a file if it has been given a STANDIN or STANDOUT attribute.

Specific Medium Attribute

Function:

The specific medium attribute supplies information concerning the nature of input/output media that are used for files at execution time.

General form:

MEDIUM (option-list)

Rules:

1. The "option list" will be defined individually for each implementation of PL/I.
2. Device-dependent information such as type, parity, density, and implementation-defined symbolic device names may be specified.

The USAGE Attribute

Function:

The USAGE attribute specifies the desired destination, such as punched cards or printed pages, for the data of output files at execution time.

General format:

USAGE (option-list)

Rules:

The "option list" will be individually defined for each implementation of PL/I.

The Storage Equivalence Attribute

Function:

The storage equivalence attribute is used to specify the sharing of storage used

for transmission to external media by two files.

General format:

```
POOL (file-name)
```

Rules:

1. The "file name" specifies a file that may share the storage area necessary for transmission to an external medium with the file currently being described.
2. It is the responsibility of the programmer to avoid confusion of usage.

The BLOCK Attribute

Function:

The BLOCK attribute specifies blocking details.

General format:

```
        (VARIABLE)
BLOCK ( {FIXED   }  ,maximum-length,
        (SPECIAL )    blocking-factor)
```

Rules:

1. Records in a block are specified as being of FIXED length, VARIABLE length, or making use of a SPECIAL technique for determining length.
2. "Maximum length" is a decimal integer constant and specifies the number of characters in the record for fixed-length records and the maximum number of characters in the record for varying-length records.
3. "Blocking-factor" is a decimal integer constant and specifies the number of records per block.

The GROUP Attribute

Function:

The GROUP attribute specifies a number of records to be treated as a group.

General format:

```
GROUP (number-of-records)
```

Rules:

The "number of records" per group is specified by a decimal integer constant.

Disposition Attributes

Function:

The disposition attributes specify the final disposition of data.

General format:

```
DISCARD | KEEP
```

Reference:

See "The OPEN Statement" and "The CLOSE Statement" in Chapter 6, for a complete explanation of these attributes.

The Function Attributes

Function:

The function attributes specify the function of a file.

General format:

```
INPUT | OUTPUT | INOUT
```

Rules:

The particular attribute used specifies that the file is to be used for input, output, or for direct-access replacement for both input and output.

File Organization Attributes

Function:

The file organization attributes, which are used in conjunction with the access attributes, specify the manner in which the records comprising a file are located.

General format:

```
( CONSECUTIVE                                  )
( REGIONAL (maximum-number-of-records)         )
( INDEXED                                      )
( COMMUNICATIONS                               )
```

Rules:

1. CONSECUTIVE specifies that the location of records within the file is dependent upon the current physical position of the device containing the file.
2. REGIONAL specifies that the location of records within the file is determined by relative physical regions of

the device containing the file. The "maximum number of records" indicates the maximum number of records within a region and must be a decimal integer constant.

3. INDEXED specifies that the location of records within a file is determined by means of an ordered index that addresses areas of the device containing the file.
4. COMMUNICATIONS specifies that records are obtained from the file in a sequence determined by a queue.

Access Attributes

Function:

The access attributes specify the manner in which the records within a file are accessed.

General format:

SEQUENTIAL|DIRECT

Rules:

1. SEQUENTIAL specifies that the next record to be accessed is determined by the order implicitly given the file by virtue of its organization.
2. DIRECT specifies that the next record to be accessed is determined by an explicitly stated identification (see the KEY, NEWKEY, and REGION options with the READ and WRITE statements, in Chapter 6). Direct access is permitted only on files organized in the regional or indexed fashion.

The KEYLENGTH Attribute

Function:

The KEYLENGTH attribute specifies the length, in characters, of the keys associated with records within files organized in the regional or indexed mode.

General format:

KEYLENGTH (length)

Rules:

For indexed organization, this attribute must appear.

Default:

For regional organization, absence of this attribute implies a key length of zero.

The ZERO Attribute

Function:

The ZERO attribute specifies that trailing blank characters in data input fields read by format items E, F, or G are to be treated as numeric zeros.

General format:

ZERO

Rules:

The ZERO attribute has no effect on output data.

The OPTIONS Attribute

Function:

The OPTIONS attribute specifies, in the file declaration, the use of various options with the input/output statements.

General format:

OPTIONS (option [,option]...)

Rules:

1. This attribute will be required only for those implementations that require information, within the file declaration, about the various ways the input/output facilities are employed.
2. In the general format, "option" is defined as attributes and options that may appear in various input/output statements.

RELATIONSHIP OF DECLARED, IMPLICIT, AND DEFAULT ATTRIBUTES

The attributes of names may be explicitly declared, deduced from context, or be assigned, as necessary, by default.

EXPLICIT ATTRIBUTES

Explicitly declared names first are classified by analyzing all DECLARE statements, statement labels, and entry labels, using the rules of scope, to distinguish among the different uses of the same identifier.

IMPLICIT ATTRIBUTES

Following the analysis of DECLARE statements, the contextual usage of identifiers is examined. Identifiers occurring in cases where only file names are allowed are classified as file names. Following is a list of such cases:

FILE option (READ, WRITE, SPACE, GROUP, SKIP, PAGE, LAYOUT, SORT)
File specification (OPEN, CLOSE)
GIVING option (SORT)
POOL option (DECLARE)
File conditions (ON, REVERT)
The built-in function POINT
FROM option

Identifiers occurring in statements of the following form are classified as external entry names:

```
CALL identifier [argument-list];
```

However, if the occurrence of the entry name lies within the scope of the same name used to label a PROCEDURE or ENTRY statement, the name is classified as an internal entry name. Identifiers appearing in CALL options are similarly treated.

Identifiers occurring in cases where only task identifiers are allowed (TASK option, WAIT statement, COMPLETE built-in function) are classified as task identifiers.

Names that are used as task identifiers are EXTERNAL, but they may not be declared in the program with the EXTERNAL attribute. However, they are known only in the attaching task and may only be used there.

Only the foregoing contextual uses are examined. Contextual uses of an identifier must be consistent with one another and with the explicit declaration, if any, within the same scope. When there is no explicit declaration for a name, the scope is assumed to be the entire external procedure.

Default attributes, as modified by IMPLICIT statements, are then given to all names. However, those names which have been classified as described above are not given attributes that would conflict with that classification.

Finally, if an identifier appears in an expression with an immediately following parenthesized list, and that identifier has no dimension attribute, it is further classified as an entry name. The classification as an external or internal entry name is made in the same manner as for names occurring in the CALL statement. This classification by usage as a function reference must not conflict with the attributes already attached to the name as specified in the preceding paragraphs.

DEFAULT ATTRIBUTES

The following is a summary of the default characteristics assumed by the name associated with the various types of attributes.

Data Type: If none of the attributes CHARACTER, BIT, or LABEL is specified, arithmetic data type is assumed.

Scalar Variables: If the range of the values is not specified, it is assumed that the variable may take on values over the entire set of data of that type.

Arithmetic Variables: See the "Default Conditions for Arithmetic Data."

Entry Names: If an external entry name appears as a function reference, the entry is assumed to be NORMAL; otherwise the entry is assumed to be ABNORMAL. External entry labels invoked in CALL statements and internal entry labels (however they are invoked) are assumed to be ABNORMAL.

If a procedure has multiple *entry names* and no data attributes, there is potential ambiguity in the characteristics of the value to be returned. In order to avoid this ambiguity, succeeding labels are interpreted as if they were entry names for successive ENTRY statements. For example, in the following, statement 1 is interpreted as if both statement 2 and statement 3 had been written.

```
1.  A: B: ENTRY;
2.  A: ENTRY;
3.  B: ENTRY;
```

File Names: The EXTERNAL attribute is assumed.

Scope: If the scope is unspecified for variable names, INTERNAL is assumed. If scope is unspecified for file names, EXTERNAL is assumed.

Storage Class Attributes: If EXTERNAL scope is declared and the storage class is unspecified, STATIC is assumed. If INTERNAL scope is declared and the storage class is unspecified, or if neither storage class nor scope is specified, AUTOMATIC is assumed.

Label Variables: The range of a label variable is assumed to be all statement labels known within the scope of the variable.

Symbol Table Attributes: If no symbol table attribute is specified, NOSYMBOL is assumed unless the name appears in the list for data-directed input or output.

Abnormality: If none of the attributes ABNORMAL, NORMAL, USES, or SETS is given, an entry name is assumed to be normal when invoked as an external function; otherwise it is assumed to be abnormal.

If ABNORMAL or NORMAL attributes are not given to data, the data is assumed to be NORMAL.

File Description Attributes: If any of the file description attributes is given, FILE is assumed.

Input/Output: If, before INPUT or OUTPUT is established, any reference is made to a file in a GROUP, SPACE, SKIP, or SEGMENT statement, the default assumption is INPUT.

STRUCTURE DECLARATIONS AND ATTRIBUTES

This section is a summarization of data declarations and attributes as they apply specifically to structures.

LEVEL NUMBER

The outermost structure is a major structure, and all contained structures are minor structures.

A structure is specified by declaring the major structure name and, following it, the names of all contained elements. Each name is preceded by a level number. A major structure is always at level one and all elements contained in a structure (at level n) have a level number that is numerically greater than n, but they need not necessarily be at level n+1.

A minor structure at level n contains all items with level numbers greater than n that are declared before the next item with a level number less than or equal to n is declared. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a declaration list.

STRUCTURES AND THE DIMENSION ATTRIBUTE

When a structure name is given the dimension attribute, it is an array of structures, and all contained items are arrays or arrays of structure (see "Arrays of Structures," in Chapter 2). Contained scalar items, contained structure elements, and cross sections of contained arrays are referred to, respectively, by subscripted names, subscripted qualified names, and the asterisk notation (see "Naming," in Chapter 2).

STRUCTURES AND DATA ATTRIBUTES

Structure names may not be given data attributes. Data attributes factored from regions containing structure declarations are not taken to apply to the structures.

STRUCTURES AND SCOPE ATTRIBUTES

Major structure names may be declared with the EXTERNAL attribute. Items contained in structures may not be declared with the EXTERNAL attribute, and even if INTERNAL is unspecified, they are assumed to be INTERNAL.

STRUCTURES AND STORAGE CLASS ATTRIBUTES

All items in the same structure must be of the same storage class; therefore, only the major structure may be given a storage-class attribute. The storage class of the major structure applies to all elements of the structure. If a structure has the CONTROLLED attribute, only the major structure, not its elements, may be allocated and freed.

EXPRESSIONS

There are three types of expressions: scalar, array, and structure. A scalar expression returns a scalar result; an array expression, an array result; a structure expression, a structure result. An expression may consist of a single operand, an operand preceded by an operator (a prefix operator), or two operands connected by an operator (an infix operator). An expression may be enclosed in parentheses to indicate that it is a single operand.

If the characteristics of the operands are different, the necessary conversion is performed before evaluation.

SCALAR EXPRESSIONS

A scalar expression is any of the following:

- a constant
- a scalar variable
- a function reference
- an expression enclosed in parentheses
- any expression preceded by a prefix operator
- any two expressions connected by an infix operator

A scalar expression returns a scalar value. The type of the value is the type of the expression. The type of the expression is dependent upon the class of operators -- arithmetic, comparison, bit string, and concatenation (see "Operators").

If A and B are expressions, then the operators + and - used in expressions of the form +A or -A, are called prefix operators. When these operators are used in expressions of the form A+B or A-B they are called infix operators.

Arithmetic Operations

An arithmetic expression of any complexity is composed of a set of elementary arithmetic operations.

An elementary arithmetic operation has the general form:

```
{{+|-} operand} | {operand
   {+| - | * | / | **} operand}
```

The general form specifies the prefix operations of plus and minus and the infix operations of addition, subtraction, multiplication, division, and exponentiation. Operands must refer to arithmetic data. If necessary, the data will be converted to arithmetic type before the operation is performed.

Mixed Characteristics

The two operands of an arithmetic operation may differ in base, scale, mode, and precision. When they differ, conversion takes place according to the following rules:

BASE: If the bases differ, the decimal operand is converted to binary, and the result is binary.

SCALE: If the scales of the operands differ and the operation is not exponentiation, the fixed-point operand will be converted to floating point. The result will be floating point. If the operation is exponentiation and the second operand is a positive integer constant, the operation will be treated as repeated multiplication until the exponent is too large. The scale of the result will be the scale of the first operand. If the operation is exponentiation and the second operand is not a positive integer constant, the result will be floating point and will be found by an approximation method; the precision of the approximation method will be the precision of the first operand.

MODE: If either operand is complex, neither operand will be converted. In the arithmetic operation, the real number will be treated as a complex number having the real number as the real part and zero as the imaginary part. The result will be complex.

PRECISION: This conversion depends upon the scale of the operands.

Floating Point: Precision is defined for floating-point numbers as the number of digits carried in the representation of the fraction. The precision of a floating-point result will be the greater of the precisions of the two operands.

Fixed Point: When performing a series of arithmetic operations, the programmer must take into consideration the fact that it may be necessary to adjust intermediate results (see the built-in functions TRUNC and ROUND, in Appendix 1) in order to prevent an overflow condition. This is particularly true in the case of division.

In the rules for fixed-point precision, the following symbols are used:

N	the length of the largest number in the implementation
m	the total number of positions in the result
n	the scale factor of the result
p	the total number of positions in operand one
q	the scale factor of operand one
r	the total number of positions in operand two
s	the scale factor of operand two
t	the computed total number of positions in the result
u	the computed scale factor of the result

Note: See "The Precision Attribute" for a detailed description of total number of positions and scale factor (p and q). The scale of the result is assumed to be FIXED.

Addition and Subtraction:

Rule:

t = max (p-q, r-s) +u+1
u = max (q,s)

Example:

```
DECLARE A FIXED (5,3), B FIXED (4,1);

    A+B
```

t = max (5-3, 4-1) + 3 + 1 = max (2,3) + 4 = 7
u = max (3,1) = 3

The result is a seven-position number with three places to the right of the decimal point.

Multiplication:

Rule:

t = p+r
u = q+s

Example:

(The fields A and B are as defined in the previous example.)

A * B

t = 5+4 = 9
u = 3+1 = 4

The result is a nine-position field with four places to the right of the decimal point.

Division:

Rule:

t = N
u = N - p+q - s

Exponentiation:

Rule:

t = p * value (y)
u = q * value (y)

In the above, y is a positive integer constant. However, if (p * value (y)) >N or if y is not a positive integer constant, conversion to floating point occurs.

Example:

```
DECLARE A FIXED (4,3);
A ** 3
```

Substituting the defined values:

t = 4 * 3 = 12
u = 3 * 3 = 9

The result is a twelve-position field with nine positions to the right of the decimal point.

The precision (m,n) is then defined as follows:

If t≤N, then m = t, n = u
If t>N, then m = N, n = u

In the latter case, truncation is performed on the left.

Conversion from floating-point scale to fixed-point scale will occur only when a destination precision is known. The destination precision will define the conversion precision.

Arithmetic Mode Conversion

If a complex value is converted to a real value, the result is the real part of the complex value.

If a real value is converted to a complex value, the result is a complex value that has the real value as the real part and zero as the imaginary part.

Table 2. Arithmetic Base and Scale Conversion

	Before Conversion			
After	Binary Fixed (p,q)	Decimal Fixed (p,q)	Binary Float (p)	Decimal Float (p)
Binary Fixed	(p,q)	CEIL (p*3.32), CEIL (q*3.32)		
Decimal Fixed	CEIL (p/3.32), (CEIL (q/3.32))	(p,q)		
Binary Float	(p)	(CEIL (p*3.32))	(p)	(CEIL (p*3.32))
Decimal Float	(CEIL (p/3.32))	(p)	(CEIL (p/3.32))	(p)

Integer Conversion

If conversion to integer is specified, as in the evaluation of subscript expressions, the conversion will be to fixed-point binary (x,o). Here x is the total number of positions in the field and depends upon the implementation. The scale factor is zero. Truncation, if necessary, will be toward zero.

Arithmetic Base and Scale Conversion

Table 2 defines the precision resulting from base and scale conversion. CEIL refers to the ceiling of the expression. (The "ceiling" of a number is the smallest integer equal to or greater than the number.)

Example:

If a binary fixed field is converted to a decimal fixed field, the precision is determined by the following method:

1. Dividing the total number of positions in the field (p) by the constant 3.32 and taking the ceiling of the quotient. This gives the precision of the new p.
2. Dividing the scale factor (q) by the constant 3.32 and taking the ceiling of the quotient. This gives the new scale factor.

Bit-String Operations

Bit-string operations have the following general forms:

```
¬ operand
operand & operand
operand | operand
```

The operations "not," "and," and "or" are specified above. The operands will be converted to bit-string type before the operation is performed. The result will be of bit type. If the operands are of different lengths after conversion, the shorter is extended on the right with zeros to the length of the longer. The length of the result will be of this extended length.

The operations are performed on a bit-by-bit basis. As a result of the operations, each bit portion has the value defined in the following table:

A	B	NOT A	NOT B	A AND B	A OR B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Examples:

If field A is '010111'B, field B is '111111'B, and field C is '101'B, then

```
       ¬ A yields '101000'B
     C & B yields '101000'B
   A | ¬ C yields '010111'B
¬ A | ¬ C & B yields '111000'B
```

For a discussion of how these expres-

sions are evaluated, see "Evaluation of Expressions," in this chapter.

Comparison Operations

Comparison operations have the general form:

operand {<|<=|=|¬=|>=|>} operand

There are three types of comparisons:

1. Algebraic, which involves the comparison of signed numeric values.
2. Character, which involves left-to-right, pair-by-pair comparisons of characters according to a collating sequence. If the operands are of different length, the shorter is extended to the right with blanks.
3. Bit, which involves the left-to-right comparison of binary digits. If the strings are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison is a bit string of length one; the value is '1'B if the relationship is true or '0'B if it is false.

If the operands of a comparison are of different types, the operand of the lower type is converted to the operand of the higher type. The priority of types is (1) arithmetic, (2) character string, (3) bit string.

As a result of the conversion, both operands will then be arithmetic or character string, and algebraic or character comparison will be performed.

Concatenation Operations

Concatenation operations have the following general form:

operand||operand

If both operands are of bit-string type, no conversion is performed, and the result is of bit type. In all other cases, the operands are converted where necessary to character-string type before the concatenation is performed, and the result is of character type.

Examples:

If A is '010111'B, B is '101'B, C is 'XY,Z' and D is 'AA/BB', then

```
   A||B yields '010111101'B
A||A||B yields '010111010111101'B
   C||D yields 'XY,ZAA/BB'
   D||C yields 'AA/BBXY,Z'
```

Type Conversion

Bit String to Character String

The bit 1 becomes the character 1, and the bit 0, the character 0. The length is unchanged.

Character String to Bit String

The characters 1 and 0 become the bits 1 and 0. The conversion is illegal if the character string contains characters other than 0 and 1.

Character String to Arithmetic

The character string is interpreted according to the rules of list directed input, i.e., the contents of the string must be a valid constant, with optional sign prefixed and with optional surrounding blanks. The value is converted directly to an operand with the same base, scale, mode, and precision that a decimal real fixed-point variable of default precision would have been converted to if it had appeared.

Bit String to Arithmetic

The bit string is interpreted as an unsigned binary integer, and converted to binary fixed, precision (S,0), where S depends upon the implementation.

Arithmetic to Character String

CODED ARITHMETIC AND BINARY NUMERIC FIELDS: The arithmetic value is converted to a character string according to the rules of list directed output specified in Chapter 5.

DECIMAL NUMERIC FIELDS: The numeric field is interpreted as a character string (see Appendix 2).

Arithmetic to Bit String

CODED ARITHMETIC AND DECIMAL NUMERIC FIELDS: The magnitude of the arithmetic value is converted to binary fixed, precision (p,0), where p is related to the

precision before conversion as follows (with ceilings of expressions used):

BINARY FIXED (r,s)	p = max(r-s,o)
BINARY FLOAT (r)	p = (r)
DECIMAL FIXED (r,s)	p = max((r-s)*3.32,o)
DECIMAL FLOAT (r)	p = (r*3.32)

The resulting binary fixed-point value is interpreted as a bit string of length p.

BINARY NUMERIC FIELDS: The numeric field is interpreted as a bit string.

ARRAY EXPRESSIONS

If the operands of an expression refer to arrays or to a combination of scalars and arrays, the expression is an array expression.

An array expression returns an array result. That is, all operations performed on arrays are performed on an element-by-element basis. Therefore, all arrays referred to in an array expression must be of identical bounds.

Note: Array expressions are not always expressions of conventional matrix algebra.

The appearance of a function reference (other than a built-in function) will imply a scalar result. For example, if A is an array, PROCEDURE (A) is a scalar function with an array argument.

Prefix Operators and Arrays

The result of the operation of a prefix operator or a built-in function upon an array is an array of identical bounds, each element of which is the result of the operation having been performed upon each of the corresponding elements of the original array.

Example:

```
If A is the array         5   3  -9
                          1  -2   7
                          6   3  -4

then -A is the array     -5  -3   9
                         -1   2  -7
                         -6  -3   4
```

Infix Operators and Arrays

Scalar - Array Operations

The result of an operation in which a scalar and an array are connected by an infix operator is an array of bounds identical to the original, each element of which is the result of the operation performed upon the scalar and upon each of the corresponding elements of the original array.

Example:

```
If A is the array          5  10   8
                          12  11   3

then 3*A is the array     15  30  24
                          36  33   9
```

Array - Array Operations

The result of an operation in which two arrays of identical bounds are connected by an infix operator is an array of bounds identical to the original arrays, each element of which is the result of the operation performed upon the corresponding elements of the two original arrays by the infix operator.

Example:

```
If A is the array         2   4
                          3   6
                          1   7
                          4   8

and if B is the array     1   5
                          7   8
                          3   4
                          6   3

then A + B is the array   3   9
                         10  14
                          4  11
                         10  11

and A*B is the array      2  20
                         21  48
                          3  28
                         24  24
```

Array Expressions Involving Structures

An array expression may involve an array of structures.

Example:

If A and B are arrays of structures:

```
1 A(3)     1 B(3)
  2 P        2 C
    3 Q        3 D
    3 R        3 E
  2 S        2 F
    3 T        3 G
    3 U        3 H
```

Then, A+B*2 is a valid expression that will result in each element of the array B being multiplied by the constant 2 and added to the corresponding element of the array A. The above expression, A+B*2, is equivalent to the following:

```
A(1).P.Q+B(1).C.D*2,
A(1).P.R.+B(1).C.E*2,
.
.
.
.
A(3).S.U+B(3).F.H*2
```

STRUCTURE EXPRESSIONS

The operands of a structure expression are structures, or a combination of structures and scalars. A structure expression returns a structure result. Array operands are not allowed in structure expressions. Note that the term "expression," as used in this specification, does not include array expressions.

All operations performed on structures are performed on an element-by-element basis. Thus, all structures appearing in a structure expression must have identical structuring. This means that the structure must have the same number of contained scalars and arrays. The positioning of the scalars and arrays within the structure must be the same, and arrays similarly positioned must have identical dimensions and bounds. The data types need not be the same.

When an operation has one structure and one scalar operand, it is interpreted as many operations, one for each scalar element in the structure. Each sub-operation involves a structure element and the scalar operand.

A structure expression is a shorthand method of applying an expression to each item of a structure.

Example:

If there are two structures:

```
1 A                   1 B
2 PART1               2 PART1
  3 SUBPART1            3 SUBPART1
  3 SUBPART2            3 ALPHA
  3 SUBPART3            3 SUBPART2

2 PART2               2 PART2
  3 SUBPART4            3 ALPHA
  3 BETA                3 SUBPART4
  3 SUBPART5  (3)       3 SUBPART5  (3)
```

Then the expression A-2*B is shorthand for the following expressions:

```
A . SUBPART1 - 2*B . SUBPART1
A . SUBPART2 - 2*B . PART1 . ALPHA
A . SUBPART3 - 2*B . SUBPART2
A . SUBPART4 - 2*B . PART2 . ALPHA
A . BETA - 2*B . SUBPART4
A . SUBPART5 - 2*B . SUBPART5
```

Note that the last expression is an array expression.

EVALUATION OF EXPRESSIONS

In the evaluation of expressions, the priority of operations is as follows:

```
**, prefix +, prefix - highest
*,/                         |
infix +, infix -            |
> =, >, ¬ =, <, < =, =      |
¬                           |
&                           |
|                           V
||                       lowest
```

Operations within an expression are performed conceptually in order of decreasing priority. For example, in the expression A + B**3, exponentiation is effectively performed before addition. If an expression involves operations of the same priority, the operations **, prefix +, and prefix - are performed from right to left and all other operations are performed from left to right.

If an expression is enclosed in parentheses, it is treated as a single operand. The parenthesized expression is evaluated before the operation of which it is an operand is performed. If both operands of an operator are parenthesized expressions, the left parenthesized expression will be evaluated first, the right parenthesized expression next and the operation involving both operands last. For example, in the expression (A + B**3)/(C*D||E), the operation will be performed in the following order:

```
B**3, which yields F
A + F, which yields G
C*D, which yields H
H is concatenated with E, which yields
  I
G/I
```

Thus, parentheses modify the normal rules of priority.

Although the infix operators + and * are associative, low-order rounding errors will depend upon the order of evaluation of an expression. For example, (A + B + C) will not necessarily produce the same result as (A + C + B).

The rules relating to abnormal functions and abnormal data should be noted (see "Abnormality," in Chapter 8).

ORDER OF THE EVALUATION OF EXPRESSIONS

Consider any scalar expression in which two operands are separated by an infix operator in the form of A op B, where "op" denotes any operator. Then either A or B, or both, may be a "composite operand", that is, a subscripted name, a function reference, or a subexpression of the form op C, C op D, or (C). In such cases, the subscripts and arguments that must be evaluated and the functions that must be invoked before the operator is applied, are termed the "elements" of the operand. For the purposes of this discussion, an operand that is an unsubscripted name or a constant is termed a "simple operand."

If A is a simple operand and B is not a simple operand, then A will not be accessed until all the elements of B are accessed. Subscript lists are evaluated and accessed, left to right, immediately before the accessing of the array elements. Argument lists are evaluated and accessed, left to right, immediately before the value of the function is accessed, i.e., immediately before the function is invoked.

Array expressions are evaluated by performing, in turn, a complete scalar evaluation of the expression for each position of the array. The evaluations proceed in row-major order. The result of an evaluation for an earlier position can alter the result for the evaluation of a later position (see Example 1, for "The Assignment Statement," in Chapter 6).

Structure expressions are evaluated by performing a complete scalar evaluation of the expression for each eligible field, in the order in which the fields in the structures are declared. The results of an evaluation for an earlier position can alter the result for the evaluation of a later position.

Processing of data consists of three operations: acquisition of data, data manipulation, and disposition of data. Input/output activity deals with the acquisition and disposition of data.

All input/output activity is performed with named collections of data called files. The name of a file is a file name. Files may be subdivided into smaller collections of data called records. Furthermore, records may be ordered within a file so that the data conceptually constitutes a single stream upon which the record structure has been superimposed. Records in such ordered files may be collected into groups.

The natural record structure of a data stream may be inappropriate to some applications. For such applications, significant divisions of data may be indicated by arbitrary symbols called segment marks. Such divisions are called segments. A record boundary within a segment may be disregarded entirely. Record boundaries may be made significant and insignificant, in turn, within one data stream.

Normally, data is transmitted between the external medium and internal storage as a record. The record may be considered a continuous string of characters or bits, with the string subdivided into contiguous substrings called fields. A field may be empty, or it may contain one and only one item of data.

The number of fields in a record, the size of those fields, the nature of the data item in each field, and the segment marks, if any, is called the format of the record. The order of items is specified by a list of elements. On input, the elements are variables or pseudo-variables to which are assigned the values of the corresponding items of data. On output, the elements are expressions whose values are given to the corresponding items of output data. Data is transmitted as if a field pointer moved across the record in synchronization with the processing of the list elements. The positioning of the pointer is governed by format items and positioning statements given for the record. The list element and the format may be specified in the record or may be specified in a list of elements and a format specification in the program.

DATA TRANSMISSION

There are four modes of data transmission: list-directed, data-directed, format-directed, and procedure-directed.

LIST-DIRECTED TRANSMISSION

List-directed transmission permits the user to specify the storage area to which data is assigned or from which data is transmitted.

Input: The data on the external medium is in the form of valid constants or expressions to represent complex constants. The program storage areas to which the data is to be assigned is specified by a data list.

Output: The data values to be transmitted are specified by a data list. The form of the data on the external medium is a function of the data value.

DATA-DIRECTED TRANSMISSION

Data-directed transmission permits the user to read or write self-identifying data.

Input: The data on the external medium is in the form of valid constants and includes information defining the program storage areas to which the data is to be assigned.

Output: The data values to be transmitted are specified by a data list. The data on the external medium has the form of valid constants and includes the name of the data being transmitted.

FORMAT-DIRECTED TRANSMISSION

Format-directed transmission permits the user to specify both the storage area to which data is to be assigned or from which data is to be transmitted and the manner of conversion of these data items.

Input: The form of the data on the external medium is defined by a format list.

The program storage areas to which the data is to be assigned is specified by a data list.

Output: The data values to be transmitted are defined by a data list. The form that the data is to have on the external medium is defined by a format list.

PROCEDURE-DIRECTED TRANSMISSION

Procedure-directed transmission provides the ability to process each data field during transmission by invoking a procedure with a CALL option.

When a procedure is invoked by a CALL option in a procedure-directed data specification in a READ or WRITE statement, the file specified in that statement becomes the current file. Such a file no longer is the current file upon return to the invoking procedure.

If a READ or WRITE statement, appearing in the invoked procedure, in turn invokes a procedure, etc., a stack of current files is created. Thus, the current file is chosen from the top of a stack. When there already is a current file immediately prior to execution of a READ or WRITE statement, that file becomes current again at the completion of the READ or WRITE statement.

This concept of a current file makes possible the use of GET and PUT and various positioning statements without having to specify the file name in the statements.

A GET or PUT statement that is executed when there is no current file is in error and causes job termination.

Examples:

```
1.  Z1:  PROCEDURE;
         DECLARE FILEX FILE;
         .
         .
         .
         READ FILE (FILEX), LIST (A,B,C),
           CALL Z2;
         .
         .
         .
         END Z1;
    Z2:  PROCEDURE;
         DECLARE FILEY FILE;
         .
         .
         .
         READ FILE (FILEY), DATA, CALL
           Z3;
         .
         .
         .
         END Z2;
    Z3:  PROCEDURE;
         DECLARE FILEZ FILE;
         .
         .
         .
         WRITE FILE (FILEZ), LIST (P,Q);
         .
         .
         .
         END Z3;
```

Upon execution of the READ statement in procedure Z1, FILEX becomes the current file. Since this READ statement invokes procedure Z2, the READ statement in Z2 is executed. This causes FILEY to become the current file. Now there are two files in the stack of current files (FILEX and FILEY). Similarly, the READ statement in Z2 invokes procedure Z3 in which FILEZ is added to the stack of current files. This stack is ordered from most current to least current as follows:

```
FILEZ
FILEY
FILEX
```

As control is returned from Z3 to Z2, FILEZ is removed from the stack of current files. Similarly, as control is returned from Z2 to Z1, FILEY is removed from the stack. Upon completion of the READ statement in Z1, there are no more current files.

```
2.  A:   PROCEDURE;
         DECLARE P FILE;
         .
         .
         .
         READ FILE (P), (D,E) (2F(8,3)),
           CALL B;
         .
         .
         .
         END A;
    B:   PROCEDURE;
         SKIP (4);
         GET LIST (F);
         .
         .
         .
         END B;
```

The READ statement in procedure A causes file P to become the current file, data items D and E to be transmitted, and procedure B to be invoked. Procedure B causes a skip to the fourth record of the current group on the current file (P), and causes data item F to be read.

DATA SPECIFICATIONS

Data specifications are given in READ, WRITE, GET, and PUT statements to indicate the relationship between the data as it is on the external medium and as it is in internal storage. The data specifications correspond to the modes of transmission.

DATA LISTS

List-directed, data-directed, and format-directed data specifications require a data list to specify the data items to be transmitted.

General format:

(element [, element] ...)

Syntax rules:

The character of the elements depends upon whether the data list is used for input or for output. The rules for each are as follows:

1. On <u>input</u>, each data-list element for format- and list-directed data may be one of the following: a scalar name, an array name, a structure name, a pseudo-variable, a pseudo-array, or a repetitive specification involving any of these elements. For a data-directed data specification, each data-list element may be an unsubscripted scalar, array, or structure name.
2. On <u>output</u>, each data-list element for format- and list-directed data specifications may be one of the following: a scalar expression, an array expression, a structure expression, or a repetitive specification involving any of these elements. For a data-directed data specification, each data-list element may be a scalar, array, or structure name, or a repetitive specification involving any of these elements.
3. Note that the elements that may be used for input are a subset of those for output.

Repetitive Specification

General format is shown in Figure 1.

Syntax rules:

1. Each element in the element list of the repetitive specification is as described for data-list elements above.
2. The expressions in the specification list are described as follows:
   a. Each expression in the specification list is a scalar expression.
   b. In the specification list, expression 1 represents the starting value of the control variable or pseudo-variable. Expression 3 represents the increment to be added to the control variable after each repetition of data-list elements in the repetitive specification. Expression 2 represents the terminating value of the control variable. Repetitions terminate as soon as the value of the control variable passes its terminating value. When the last specification is completed, control passes to the next element in the data list.
   c. If "BY expression 3" is omitted from the specification list, expression 3 is assumed to be one. If "TO expression 2" is omitted from the specification list, expression 2 is assumed to be

$$\text{(element [, element] ...} \left\{\begin{array}{l}\text{variable}\\ \text{pseudo-variable}\end{array}\right\} \text{= specification-list...)}$$

A specification list has the following format:

$$\text{expression-1} \left[\begin{array}{l}\text{TO expression-2 [BY expression-3]}\\ \text{BY expression-3 [TO expression-2]}\end{array}\right] \text{[WHILE expression-4]}$$

Figure 1. General Format for Repetitive Specification

infinite. If both "TO expression 2" and "BY expression 3" are omitted, the associated data-list elements appear only once; they are not repeated.

d. The WHILE clause specifies that before each repetition of data-list elements, the expression is evaluated and, if necessary, converted to give a bit-string value. If any bit in the resulting string has the value '1', the repetitions continue uninterrupted. If all bits have the value '0', the repetitions associated with the current specification are terminated.

3. Repetitive specification may be nested to any depth. That is, each element in the element list may be a repetitive specification. A repetitive specification involving m elements repeated n times is equivalent to m * n elements. For example, consider the following list element:

```
((A(I,J) I=1 TO 2) J=3 TO 4)
```

This gives the elements of the array A in the following order:

```
A(1,3), A(2,3), A(1,4), A(2,4)
```

Transmission of Data-List Elements

If a data-list element is of complex mode, the real part is transmitted before the imaginary part.

If a data-list element is an array name, the elements of the array are transmitted in row-major order, that is, with the rightmost subscript of the array varying most frequently.

If a data-list element is a structure name, the elements of the structure are transmitted in the order specified in the structure declaration.

If, within a data list used in an input statement, a variable is assigned a value, this new value is used in all later references in the data list.

Example:

In the following statement, B is a structure, XSTRING is a character string, and C is an array:

```
DECLARE 1B, 2P, 2E, 3F, XSTRING
   CHARACTER (6), C(10) FIXED;
```

The following data list, involving these data items, and the scalar variable A, may be used for input or output.

```
(A,B, SUBSTR (XSTRING,2), (C(I) I=1 TO
   10))
```

The data-list elements are transmitted in the following order:

A The scalar variable is transmitted.
P } The elementary elements of the
F } structure are transmitted.

SUBSTR (XSTRING, 2) The second through sixth characters of the string are transmitted.

C(1), C(2)..., C(10) The ten elements of the array are transmitted.

LIST-DIRECTED DATA SPECIFICATION

General format:

```
LIST data-list [(scalar-expression)]
```

Syntax rules:

1. The "data list" is described in the preceding discussion.
2. The "scalar expression," which may optionally follow the data list, represents a separator of data items on the external medium.

List-Directed Input

The data on the external medium is a list of constants or expressions that represent complex constants. A character constant containing a real value is valid as a constant. Thus '2.6' is a valid constant. Sterling constants are not allowed in complex data.

Where the data item is an array name and the data, scalar constants, the first constant is assigned to the first element of the array, the following constant to the second element, etc., in row-major order.

A structure name in the data list represents a list of the contained scalar variables and arrays in the order specified in the structure description.

A scalar expression, enclosed in parentheses, optionally may follow the list-directed data list. If the expression is not present, data items on the external

medium must be separated by a comma or a blank. If present, the expression is evaluated and converted, if necessary, to a character string; the resultant character string is recognized as the separator of data items on the external medium.

List-Directed Output

The values of the scalar variables in the data list are converted to a character representation of the data value, as described below, and transmitted to the external medium.

A scalar expression, enclosed in parentheses, optionally may follow the list-directed data list. If the expression is not present, a blank is used to separate data items to be transmitted. If present, the expression is evaluated and converted, if necessary, to a character string; the resultant character string is used to separate data items to be transmitted.

List-Directed Output Format

Data fields written under list direction are aligned in vertical columns. This alignment is called tabbing, and is separately defined for each implementation of the language. System tabbing may be overridden by a TAB option in a LAYOUT statement for the appropriate file.

Each item of a list-directed data specification, except the last, is immediately followed by the separating character indicated in the list-directed data specification by a scalar expression enclosed in parentheses. If the data is to be reread under list direction, care should be taken to avoid including character data or numeric fields in the output and to avoid ambiguity resulting from choice of separating or terminating characters.

Data items are written on successive tab positions unless an item extends into a succeeding position. (In that case, the next free position is used.) A data item may span several tabs, but must not span page lines or cards.

Length of List-Directed Output Fields

The length of the data field on the external medium is a function of the internal precision and value of the data item.

CODED ARITHMETIC DATA: The external form of coded arithmetic data is a valid decimal constant whose field width, w, is a function of the internal precision (p,q) declared for the data item and the value of the data item. In the discussion below, the following abbreviations are used:

1. The letter w represents the field width, which is defined as the length of the data field on the external medium.
2. The letter d represents the number of positions in the external data field to the right of the decimal point.
3. The letter p represents the total number of significant digits in the data field.
4. The letter q represents the number of significant digits to the right of the decimal point.
5. The letter z represents the total number of leading zeros to be suppressed.
6. The letter t represents the existence or nonexistence of a decimal point; t takes on the values 0 and 1 depending upon the value of q. If q is nonzero, an explicit decimal point is required on the external medium, and t has the value 1. If q is zero, implying that no digits appear to the right of the decimal point, t has the value 0. To summarize:

   if $q = 0$, then $t = 0$
   if $q \neq 0$, then $t = 1$

7. The letter u represents the existence of a minus sign; u takes on the values 0 and 1 depending on the value of the data. If the data value is less than zero, a minus sign is placed to the left of the first digit in the data field on the external medium, and u has the value 1. If the data value is equal to or greater than zero, no sign is required in the data field, and u has the value 0. To summarize:

   if data value < 0, then $u = 1$
   if data value ≥ 0, then $u = 0$

8. The letter s represents a scale factor as described for floating-point data.

There are five kinds of coded arithmetic data to consider: coded real fixed-point decimal data, coded real fixed-point binary data, coded real floating-point decimal data, coded real floating-point binary data, and coded complex data.

Coded Real Fixed-Point Decimal Data: The field width w is equal to $(p - z + t + u)$.

Coded Real Fixed-Point Binary Data: The data is converted to fixed-point decimal with a precision (p,q) and is transmitted as coded real fixed-point decimal data.

Coded Real Floating-Point Decimal Data: The data is converted according to the rules for fixed-point format items, F(w,d). For F-conversion, if p is the declared precision of the data item, w = p + 2 and d = p - 1. For list-directed output, the field containing the converted data item is padded on the right with four blank characters, such that the total field width is w + 4. The effect is similar to the pair of format items F(w,d), X(4).

If this conversion causes either a digit overflow into the sign position or a significant zero digit in the position immediately to the right of the decimal point, the data item is converted according to the rules for floating-point format items, E(w,d,s). For E-conversion, w = p + 6, d = p - 1, and s = p.

Coded Real Floating-Point Binary Data: The data is converted to floating-point decimal with a precision (p) and transmitted as coded real floating-point decimal data.

Coded Complex Data: The data is represented as two immediately adjacent real data fields, the left-hand field being the real part of the data and the right-hand field being the imaginary part of the data.

A sign always precedes the imaginary part. If the value of the imaginary part is greater than, or equal to, zero, the sign is plus; if the value of the imaginary part is less than zero, the sign is minus. The imaginary part is always followed by the letter I.

The field width of the real part of the complex data is as described in the items above, whichever applies. The field width of the imaginary part is as described above plus 2 (one position for the sign and one for the letter I) if u = 0, or plus 1 (for the letter I), if u = 1. Therefore, the field width of the complex data field is the sum of the widths of the subfields.

NUMERIC FIELD DATA: The base of numeric field data is either decimal or binary.

Numeric Decimal Data: The external format and field width of the numeric decimal data item is that described by a picture specification.

Numeric Binary Data: The external format and field width of the numeric binary data item is that described by a picture specification. The binary digits 0 and 1 are represented by the characters 0 and 1.

CHARACTER-STRING DATA: The contents of the character string are written out. Enclosing quotation marks are not supplied, and contained quotation marks are unmodified. The field width is the current length of the string.

BIT-STRING DATA: The format of the data on the external medium is that of a bit-string constant, that is, the value is enclosed in quotation marks and followed by the letter B. The field width is (p + 3), where p is the current length of the string and the three additional positions are for the two quotation marks and the letter B.

Examples of list-directed data specifications:

```
READ LIST (CARD.RATE,
   DYNAMIC_FLOW);
READ LIST ((THICKNESS (DISTANCE)
   DISTANCE = 1 TO 1000));
WRITE LIST (P, Z, M, R);
WRITE LIST (A * B / C, (X + Y)  **  2)
   (',');
```

The first two examples are list-directed input specifications and the latter are output specifications.

DATA-DIRECTED DATA SPECIFICATION

General format:

Option 1
    DATA

Option 2
    DATA data-list

Syntax rules:

1. The data list is described in "Data Lists," in this chapter.
2. Option 1 implies that all of the data items to be transmitted are known to the block containing the READ or GET statement and are declared with a SYMBOL attribute. Option 1 may be used for data-directed input only.
3. Option 2 may be used for both data-directed input and output.

Data-Directed Data On External Medium

The data on the external medium associated with data-directed transmission is in the form of a list of scalar assignments having the following general format:

```
scalar-variable = constant [{,
                             blank}
   scalar-variable = constant]...;
```

Syntax rules:

1. The "scalar variable" may be a subscripted name with decimal integer constant subscripts, but may not be declared with the DEFINED attribute.
2. On input, the scalar assignments may be separated by either a blank or a comma. On output, the assignments are always separated by blanks.

General rules for data-directed <u>input</u>:

1. If the data specification in Option 1 is used, the names on the external medium may be any unqualified name known at the point of transmission and declared with the SYMBOL attribute.
2. If Option 2 is used, the elements of the data list may be any unsubscripted scalar, array, or structure names. The names on the external medium must appear in the data list; however, the order of the names need not be the same, and the data list may include names that do not appear on the external medium.

   For example, consider the following data list, where A, B, C, and D are names of scalar variables:

   ```
   DATA(B,A,C,D)
   ```

   This data list may be associated with the following input data stream:

   ```
   A = 2.5, B = .00476 D = 125;
   ```

   Note that the assignments for A and B are separated by a comma, while the assignments for B and D are separated by a blank. Also, C appears in the data list but not on the external medium.
3. If the data list in Option 2 includes the name of an array, subscripted references to that array may appear on the external medium. The entire array need not appear on the external medium.

   Let X be the name of a two-dimensional array declared as follows:

   ```
   DECLARE X (2,3);
   ```

   Consider the following data list and input data stream:

Data List	Input Data Stream
DATA (X)	X(1,1) = 7.95, X(2,1) = 8085, X(1,3) = 73;

   Although the data list has only the name of the array, the associated input stream may contain values for individual elements of the array.
4. If the data list includes qualified names, then qualified names of identical form may appear on the external medium. Subscripted qualified names may have interleaved subscripts in the data list but not on the external medium. Consider the following structures:

   ```
   DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRP
     SYMBOL(X), 2 PRICE SYMBOL(X1)

   1 CARDOUT, 2 PARTNO, 2 DESCRP
     SYMBOL(X2), 2 PRICE SYMBOL(X3);
   ```

   If it is desired to read a value for CARDIN.PARTNO, then the data list and input data stream have the following forms:

Data List	Input Data Stream
DATA (...CARDIN . PARTNO...)	...CARDIN . PARTNO = 737314 ...;

5. When a structure name is given in a data list, elementary structure elements, which have been declared with the SYMBOL (identifier) attribute, may be transmitted. In this case, the scalar variable in the assignment list on the external medium is the identifier given in the SYMBOL attribute. The equivalent qualified name may not appear in the assignment list unless the qualified name also appears in the data list.

   Consider the structures described in item 4. If it is desired to read values for DESCRP in CARDIN and for PRICE in CARDOUT, the following data list and input stream may be used:

Data List	Input Stream
DATA (CARDIN,CARDOUT)	X = 65, X3 = 38;

   Both of these data items may have been transmitted by using identical qualified names in both the data list and input stream as described in item 4, above.

General rules for data-directed <u>output</u>:

1. The elements of the data list may be a scalar name, an array name, a structure name, a repetitive specification involving any of these elements, or further repetitive specifications. The data with names appearing in the data list is transmitted, in the form of a list of scalar assignments separated by blanks and terminated by a semicolon.
2. Array names in the data list are treated as a list of the contained subscripted elements in row-major order.

Let X be an array declared as follows:

DECLARE X(2,4);

Let X appear in a data list as follows:

DATA (X)

Then, on output, the output data stream is as follows:

```
X(1,1)=1 X(1,2)=2 X(1,3)=3 X(1,4)=4
X(2,1)=5 X(2,2)=6 X(2,3)=7 X(2,4)=8;
```

3. Subscript expressions in a data name are evaluated and replaced by integer constants.
4. Qualified names appearing in the data list are transmitted with the same qualification, but subscripts follow the qualified name rather than being interleaved. If a data list is specified for a structure element transmitted under data-directed output as follows:

   DATA(Y(1,3).Q)

   Then the associated data field in the output stream is as follows:

   Y.Q(1,3)=3.756

5. Structure names in the data list are interpreted as a list of the contained scalar or array elements, and arrays are treated as above.

   Consider the following structure:

   1A, 2B, 2C, 3D

   If a data list for data-directed output is as follows:

   DATA (A)

   Then the associated data fields in the output stream are as follows:

   B=2 D=17;

6. Data-directed output is suitable for data-directed input only if it includes no numeric fields of binary base or numeric fields of decimal base that do not have the form of valid arithmetic constants.

Data-Directed Output Format

Data fields written under data direction are tabbed, that is, aligned in vertical columns. This tabbing may be overridden by a TAB option in a LAYOUT statement for the appropriate file.

Data items are written on successive tab positions unless an item extends into a succeeding position. (In that case, the next free position is used.) A data item may span several tabs; however, data items must not span page, lines, or cards.

Length of Data-Directed Data Fields

The length of the data field on the external medium is a function of the internal precision and the value of the data item being written. The field length for coded arithmetic data, numeric field data, and bit-string data is the same as described for list-directed output (see "Length of List-Directed Output Fields").

For character-string data the contents of the character string are written out enclosed in quotation marks. Each quotation mark contained within the character string is represented by two successive quotation marks.

Example:

Assume that A is declared as a one-dimensional array of six elements; B is a one-dimensional array of seven elements. If it is desired to calculate values for A=B (I+1) + B(I) and write them out, the procedure in Figure 2 might be used.

```
AB:   PROCEDURE;
                                          Input Stream on External Medium
      DECLARE A (6) , B(7);
                                          B(1)=1,B(2)=2,B(3)=3,
      READ DATA (B);
                                          B(4)=1,B(5)=2,B(6)=3,B(7)= 4;
      DO I = 1 to 6;

      A (I) = B(I+1) + B(I);
                                          Output Stream on External Medium
      END;
                                          A(1)=3 A(2)=5 A(3)=4 A(4)=3
      WRITE DATA (A);
                                          A(5)=5 A(6)= 7
      END AB;
```

Figure 2. Example of Data-Directed Transmission, Both Input and Output

FORMAT-DIRECTED DATA SPECIFICATION

General format:

data-list format-list

Syntax rules:

1. The data list is described in "Data Lists," the format list in "Format Lists."
2. Each data item in the data list is converted according to an associated format item in the format list: the first scalar data item with the first format item, the second scalar data item with the second format item, etc. Suppose the format list effectively contains j format items, and the data list effectively contains k data items. Then if j<k after j scalar data items have been transmitted, the format list is re-used, the (j+1)th scalar item being associated with the first format item, etc. This re-use is performed as many times as required. If j>k , redundant format items are ignored.
3. An array or a structure in a data list is equivalent to n data items, where n is the number of scalar elements in the array or structure.
4. The actions associated with the control format items encountered in the format list during transmission are performed at the appropriate point.
5. The specified transmission is complete when the last data item has been processed using the corresponding format item. Subsequent format items are ignored.

Examples:

The first of the following examples is a format-directed input specification, and the second is an output specification:

1. (NAME,DATE,SALARY) (A(COLA_COLB), X(2), A(6), F(M+2,2))
2. ('RESULT('|| I || ')=', A(I) I=1 TO 20) (A,F(8,3))

FORMAT LISTS

The format-directed data specification and the FORMAT and POSITION statements require a format list.

General format of a format list:

```
  ⎧ item              ⎫ ⎡ ,item             ⎤
( ⎨ n item            ⎬ ⎢ ,n item           ⎥...)
  ⎩ n(item [,item]...)⎭ ⎣ ,n(item [,item]...)⎦
```

Syntax rules:

1. Each "item" represents a format item as described below. Each item may have any one of these three possible forms.
2. The letter n represents an iteration factor, which is either an expression enclosed in parentheses, or a decimal integer constant. The iteration factor specifies that the associated format item is to be used n successive times. A zero or negative iteration factor specifies that the associated format item is to be skipped and not used.

If an expression is used to represent the iteration factor, it is evaluated and converted to an integer each time the associated format item is used. The associated format item is that item or list of items to the right of the iteration factor.

General rule:

There are two types of format items: data format items and control format items. Data format items specify the form of data on an external medium. Control format items specify control over records and groups being read or constructed.

Data Format Items

Data format items describe data representation in two modes: external and internal. The external mode is designed to be readable and uses character representation. The internal mode is in coded form, which is individually defined for each implementation of PL/I, and is primarily used for compact intermediate storage. Arithmetic internal mode format items, other than the numeric picture item (P), specify coded internal form. The P format item specifies that the internal form is numeric field or character string.

External Mode Format Items

The discussion of external mode format items requires the following definitions:

1. The letter w represents the length of the field in characters used by the external representation (including signs, decimal or binary points, and the letters E and B, as used in representation of constants).
2. The letter d represents the number of positions after the decimal or binary point.
3. The letter s represents the number of significant digits (binary or decimal) to appear.
4. The letter p represents a scale factor, which may be positive or negative.

The quantities w,d,s may be specified by an expression. When the format item is used, the expression is evaluated and converted to an integer.

On input, the data item in the external data field is converted to the characteristics of the list item. Rules for the conversion are given in Chapter 4.

There are seven format items associated with data in external mode: fixed-point (F), floating-point (E), complex (C), Picture specification (P), bit-string (A), character-string (A,P), and general (G).

FIXED-POINT FORMAT ITEMS: Numeric data may be described by a fixed-point format item.

General format:

Option 1.

F (w)

Option 2.

F (w,d)

Option 3.

F (w, [d] ,p)

General rules:

1. On input, the data items in the external data field are the character representation of decimal fixed-point numbers anywhere in a field of width w. In Option 1, only the integer portion of the number is accepted; if a decimal point appears, all digits after the point are ignored.

   In Option 2, both the integer and fractional parts are accepted. If no decimal point appears in the number, it is assumed to appear immediately before the last d digits (if trailing blanks are treated as zeros, they are included in the count of d digits). If a decimal point does appear, it overrides the d specification.

   In Option 3, the scale factor effectively multiplies the external data value by 10 raised to the value of p. If p is positive, the number is treated as though the decimal point appeared p places to the right of its given position. If p is negative, the data is treated as though the decimal point appeared p places to the left of its given position. The magnitude of p may be greater than the magnitude of w. The given position of the decimal point is that indicated either by an actual point, if it is given, or by d, in the absence of an actual point. If no d specification is given, only the integer part of the number is considered.
2. On output, the external data is a decimal fixed-point number, right-adjusted in a field of width w.

   In Option 1, only the integer

portion of the number is written; no decimal point appears.

In Option 2, both the integer and fractional parts of the number are written. If d is specified, a decimal point is inserted before the last d digits, and the value is appropriately positioned. Trailing blanks are supplied if the number of fractional digits is less than d.

In Option 3, the scale factor effectively multiplies the internal data value by ten raised to the power of p before it is edited into its external character representation. (The action is the same as on input.) If d is omitted, only the integer portion of the number is considered.

For all options, if the value of the number is less than zero, a minus *sign* will be prefixed to the external character representation; if it is greater than or equal to zero, no sign will appear. Therefore, the w specification must include a position for the minus sign. If the field width w is too small to contain all the significant digits -- and perhaps a decimal point and minus sign -- the FIELDOVERFLOW condition will be raised.

<u>FLOATING-POINT FORMAT ITEMS:</u> Numeric data may be described by a floating-point format item.

General format:

```
E (w,d [, s])
```

General rules:

1. On input, the data item in the external data field is an optionally signed character representation of a decimal floating-point number anywhere within field of width w.

   The external form of the number is as follows:

$$\begin{bmatrix} + \\ - \end{bmatrix} \text{ fixed-point-number } \left[\begin{Bmatrix} [E] & \pm \\ E & [\pm] \end{Bmatrix} \text{ exponent} \right]$$

   a. If there is no decimal point in the external data field, the decimal point is assumed to be before the last d digits. If there is a decimal point in the external data field, it overrides the decimal point placement specified by d.
   b. The "exponent" is a decimal integer. If the exponent and the preceding E or sign are omitted, a zero exponent is assumed.

2. On output, the data item in the external data field has the following general form:

$$\begin{Bmatrix} - \\ \text{blank} \end{Bmatrix} \text{ s-d digits . d digits E} \begin{Bmatrix} + \\ - \end{Bmatrix} \text{ exponent}$$

   a. The "exponent" is a decimal integer of <u>n</u> digits, where <u>n</u> is defined individually for each implementation. The exponent is adjusted so that the leading digit of the characteristic is nonzero.
   b. If the above form does not fill the field of width w, it is right-adjusted, and blanks are inserted on the left. If s is omitted it is taken as equal to d. The field width w must be greater than or equal to (s + n + 3) for non-negative values and (s + n + 4) for negative values of the data item.

<u>COMPLEX FORMAT ITEMS:</u> Complex numeric data may be described by a complex format item.

General format:

```
C (real-format-item
   [,real-format-item])
```

General rules:

1. Each "real format item" is either a fixed-point, floating-point, picture, or general format item.
2. On input, the external data is the real and imaginary parts of the complex number in adjacent fields described by the two contained format items. If the second real format item is omitted, it is assumed to be the same as the first.
3. On output, the form of the real and imaginary parts is specified by enclosed real format items.

<u>PICTURE FORMAT ITEM:</u> Numeric data may be described by a numeric picture using the P format item.

General format:

```
P 'numeric-picture-specification'
```

The "numeric picture specification" is described in "The PICTURE Attribute," in Chapter 3.

<u>BIT-STRING FORMAT ITEMS:</u> The bit-string item describes the external representation of a bit string using the characters 0 and 1.

General format:

A [(w)]

General rules:

1. If w is omitted, it is taken to be the maximum length of the associated data list element on input or the current length on output.
2. On input, the external data is a character representation of bit string anywhere within the field of width w.
3. On output, the character representation of bit string is left-adjusted in the field of width w. Truncation, if necessary, is performed on the right. Blanks are used for padding.

CHARACTER-STRING FORMAT ITEMS: Character data may be described by a character-string format item.

General format:

$\left\{\begin{array}{l}\text{A [(w)]}\\ \text{P 'character-picture-specification'}\end{array}\right\}$

General rules:

1. The "character picture specification" is described in "The String Attributes," in Chapter 3.
2. The external representation is a string of w characters.
3. On output, truncation, if necessary, is performed on the right. If the associated list element is too short, it is extended on the right with blanks. If the picture form is used, w is implied. Checking and editing are performed.
4. If w is omitted, it is taken to be the maximum length of the associated data list element on input or the current length on output.

GENERAL FORMAT ITEMS: Both character data and numeric data may be described by the general format item.

General format:

G {(w) | (w,d) | (w, [d] ,s)}

General rules:

1. The type of the external character representation of the data is assumed to be that of the associated data-list element.
2. In the case of strings, the effect of the general format item is identical to A (w); d and s, if specified, are ignored. Coded bit-string external representation may not be described by a general format item.
3. On input for arithmetic data, the scale of the external character representation is deduced. The effect of the general format item is then identical to F (w), F (w,d) for fixed-point numbers and E (w,d,s) for floating-point numbers.

   On output for arithmetic data, the data is analysed with respect to the specified field width w.
4. If the data item may be represented without loss of accuracy as a fixed-point number, the external form is that specified by F (w), or F (w,d) if d is specified. If the data item cannot be suitably represented by an F format item, it is necessary that d be specified in the general format item. The effect is then identical to E (w,d) or E (w,d,s) if s is specified.

Internal Mode Format Items

Internal mode format items may specify precision and length. This is given in exactly the same way as with the precision attribute. The base of the precision is that of the format item, or, where this is indeterminate, that of the associated data item. If size or precision is omitted, it is assumed to be that of the associated data item, converted where necessary to the characteristics specified by the format item. The type, base, scale, mode, and precision of a data item may differ from its associated format item. Wherever this occurs, conversion is performed.

There are seven format items associated with data in internal mode: fixed-point (IF), floating-point (IE), complex (IC), picture (P), bit-string (B), character-string (A,P), and general (IG).

INTERNAL FIXED-POINT ITEMS: Numeric data in internal mode may be described by the internal fixed-point format item.

General format:

$\left\{\begin{array}{l}\text{IF [(precision)]}\\ \text{IFB [(precision)]}\end{array}\right\}$

The first form is used for decimal data, the second form for binary data.

INTERNAL FLOATING-POINT FORMAT ITEMS: Numeric data in internal mode may be described by the internal floating-point format item.

General format:

$\left\{\begin{array}{l}\text{IE [(precision)]}\\ \text{IEB [(precision)]}\end{array}\right\}$

The first form is used for decimal data, the second form for binary data.

INTERNAL COMPLEX FORMAT ITEMS: Complex numeric data in internal mode may be described by an internal complex format item.

General format:

IC (internal-real-format-item)

General rules:

1. The "internal real format item" is either an internal fixed-point, floating-point, picture, or general format item.
2. The internal real format item specifies the forms of both the real and the imaginary parts of a complex number.

INTERNAL PICTURE FORMAT ITEMS: Numeric data in internal mode may be described by an internal picture format item.

General format:

P 'picture-specification'

The "picture specification" is described in "The PICTURE Attribute," in Chapter 3.

INTERNAL BIT-STRING FORMAT ITEMS: The internal bit-string format item describes the internal representation of a bit string.

General format:

B [(length)]

General rules:

1. "Length" is the length of the string in bits. If omitted, it is taken as the current length of the associated bit-string list item.
2. The external representation is the coded form for a bit string. If s bits are encoded in one character, the width of the external field may be represented as follows:

   TRUNC (length-1)/s) + 1

3. On input, the coded string is interpreted as bit string and is truncated, if necessary, to the specification length.
4. On output, the string is extended with zeros to length TRUNC ((length-1)/s)*s+s and the external form is this string.

INTERNAL CHARACTER-STRING FORMAT ITEMS: Character data in internal mode may be described by an internal character-string format item.

General format:

$$\left\{\begin{array}{l}\text{A [(w)]}\\ \text{P 'character-picture-specification'}\end{array}\right\}$$

The "character picture specification" is described in "The Picture Attribute," in Chapter 3.

INTERNAL GENERAL FORMAT ITEMS: Both character data and numeric data in internal mode may be described by an internal general format item.

General format:

IG

The IG format item specifies that the format of the data on the external medium is to be identical to its internal form.

Control Format Items

There are two types of control format items: the spacing format item, X, and the positioning format items, which effect transmission in exactly the same way as do the statements of the same names. All of these format items except POSITION, are for use only with data in the external mode.

Spacing Format Item

General format:

X (w)

General rules:

1. On input, the format item specifies that the next w characters of the external data are to be ignored.
2. On output, the format specifies that w characters of blanks are to be inserted into the external data.

Positioning Format Items

The positioning format items are:

$$\left\{\begin{array}{l}\text{SPACE (expression)}\\ \text{SPACE}\end{array}\right\}$$

$$\left\{\begin{array}{l}\text{SKIP (expression)}\\ \text{SKIP}\end{array}\right\}$$

$$\left\{\begin{array}{l}\text{GROUP (expression)}\\ \text{GROUP}\end{array}\right\}$$

POSITION (format list)

$$\left\{\begin{array}{l}\text{TAB}\\ \text{TAB (expression)}\end{array}\right\}$$

General rules:

1. The effect of each of these format items is identical to the statements of the same names (see the individual statements, described in Chapter 6, for a description of the action taken).
2. Only the POSITION item of the above, may be used in lists intended for internal string editing.

Remote Format Item

If it is desired to locate format items remotely from a format list, the remote format item, R, may be used.

General format:

R (statement-label-designator)

General rules:

1. The "statement-label-designator" is a label constant or a label variable that is the statement label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the remote format item.
2. The R format item and the specified FORMAT statement must be internal to the same block.

PROCEDURE-DIRECTED DATA SPECIFICATION

The data specification for procedure-directed transmission has the following general format:

CALL entry-name [(argument [,argument] ...)]

The CALL option causes the procedure whose name is "entry name" to be invoked in the same manner as the CALL statement.

The invoked procedure may perform further action on the data to be transmitted by using GET, PUT, and the positioning statements.

Whereas the data-directed, list-directed and format-directed data specifications may appear more than once in a READ, WRITE, GET, or PUT statement, a procedure-directed data specification may appear only once.

INPUT/OUTPUT STATEMENTS

The input/output statements may be classified as follows: file preparation, data specification, data transmission, positioning, report generation, and save and restore. A description of each statement is given in Chapter 6, where all PL/I statements are presented in alphabetic order.

FILE PREPARATION STATEMENTS

Before data can be transmitted between internal storage and any file, certain preparations must be made, such as checking for and ensuring the availability of the data medium or allocating appropriate programming support and storage areas. These preparations are called opening the file. Similarly, when usage of a file is completed, it must be closed, to permit release of the facilities allocated for opening and to cause proper disposition of the facilities allocated for opening and to cause proper disposition of the file. The programmer may accomplish these two actions by writing the statements OPEN and CLOSE.

DATA SPECIFICATION STATEMENT

The format of a record to be transmitted may be specified by the FORMAT statement or in the data transmission statements.

DATA TRANSMISSION STATEMENTS

The READ and WRITE statements cause the transmission of data between storage and external media. Data may be transmitted in one of four modes: data-directed, format-directed, list-directed, and procedure-directed. The GET and PUT statements cause data to be transmitted between the current file and specified variables. These statements are used in conjunction with procedure-directed data transmission. The DISPLAY statement causes a message to be transmitted from the program to the machine operator.

POSITIONING STATEMENTS

Positioning within and between records or segments may be accomplished with the

POSITION, TAB, SKIP, SPACE, GROUP, and SEGMENT statements. The first two of these are for positioning within records or segments only, and they apply only to current files. The remainder are for positioning between records, groups, or segments, and they may apply either to the current file or an explicitly designated file. The REPOSITION statement has a special use with the ON statement (see "The REPOSITION Statement").

REPORT GENERATION STATEMENTS

The PAGE and LAYOUT statements are provided to facilitate preparation of files for printing and to describe the format of print files so they may subsequently be read. The statements may, however, be used for any nonprint file. Groups and records are then considered in place of pages and lines. The statements refer explicitly (or in the case of procedure-directed transmission, implicitly) to a particular file. Each statement applies to that file until overridden by another statement of the same type. Until such statements are encountered, system standards are assumed to apply.

The execution of a PAGE or LAYOUT statement for a file destroys all options established by previously executed PAGE or LAYOUT statements for the same file.

SAVE AND RESTORE STATEMENTS

The SAVE statement causes data to be saved in auxiliary storage; the RESTORE statement restores the data to the program.

This section includes a description of each statement in the language. These descriptions are presented in alphabetic order.

To show the relationships among these statements, they are also classified into logical groups.

RELATIONSHIP OF STATEMENTS

CLASSIFICATION

Statements may be classified into the following logical groups: assignment, control data declaration, error control and debug, input/output, program structure, sorting, and storage allocation.

Assignment Statement

The assignment statement is used to evaluate expressions and to assign values to scalars, arrays, and structures.

Control Statements

The control statements alter the normal sequential flow of control through a program. The control statements are GO TO, IF, DO, CALL, RETURN, WAIT, STOP, EXIT, DELAY.

Data Declaration Statements

The data declaration statements, DECLARE and IMPLICIT, specify attributes for names and identifiers. These statements are described in "PL/I Declarations," in Chapter 3.

Error Control and Debug Statements

When an interrupt occurs during program execution, standard operating system action is taken; however, the language provides the facility to override system action on these interrupts. By using the ON statement, a programmer may specify the action to be taken when an interrupt occurs and can record the status of the program at the point of the interrupt. By using the SIGNAL statement, the programmer may initiate programmed interrupts and may simulate machine interrupts to facilitate debugging.

I/O Statements

See "Input/Output Statements," in Chapter 5, for a classification and discussion of statements used in I/O operations.

Program Structure Statements

The program structure statements are: PROCEDURE, BEGIN, END, DO, and ENTRY. The first three statements delimit the scope of declarations within a program. The ENTRY statement provides a secondary entry point for a procedure.

Sorting Statement

The SORT statement sorts and, optionally, merges records on a file.

Storage Allocation Statements

The storage allocation statements are: ALLOCATE, FREE, FETCH, and DELETE. The ALLOCATE and FREE statements allocate and free storage for variables. The FETCH and DELETE statements allocate and free storage for programs.

SEQUENCE OF CONTROL

Within a block, control normally passes sequentially from one statement to the next. However, sequential operation is modified by the following statements: CALL, END, EXIT, GO TO, PROCEDURE, RETURN, SIGNAL, and STOP.

A CALL statement passes control to the specified entry point.

An END statement, logically terminating a procedure, acts as a RETURN statement, causing control to return to the invoking procedure.

The EXIT statement causes control to leave a task; the STOP statement causes control to leave a program.

A GO TO statement causes control to transfer to the specified statement label.

A PROCEDURE statement heads a procedure. Procedures may be considered as independent blocks and are placed anywhere within an external procedure, consistent with desired identifier scopes. However, a procedure may be invoked only by a CALL statement or a function reference. Thus, control passes around a nested procedure, from the statement before a PROCEDURE statement to the statement after the appropriate END statement for the procedure.

A RETURN statement returns control from a procedure to the invoking procedures.

A SIGNAL statement causes control to pass to the statement or block specified within the associated ON statement.

The following conditions may cause sequential operation to be modified:

1. A function reference in any expression causes control to pass to the specified function procedure.
2. The occurrence of an enabled condition specified in an ON statement causes control to pass to the statement or block contained in the statement.
3. The flow of control through the IF and ON statements and through a DO group may or may not be sequential.
4. In an appropriate environment, the asynchronous execution of several operations may involve transfer of control under the influence of external occurrences.

The following example illustrates sequence of control:

```
A:  PROCEDURE;
B:  X = Y + Z;
C:  CALL D;
E:  W = P*Q;
    D:  PROCEDURE;
    G:  S = T/P;
    H:  RETURN;
    I:  END D;
J:  U = V**W;
K:  GO TO N;
    .
    .
    .
N:  END;
```

Control flows in the following order: A, B, C, D, G, H, E, J, K, N.

COMMON OPTIONS

In the statement descriptions that follow, the use of pseudo-variables and the TASK, KEY, NEWKEY, and REGION options appear frequently. In the general format for each, "expression" refers to a scalar expression.

PSEUDO-VARIABLES

The following built-in functions (see Appendix 1 for a more complete description) may be used as pseudo-variables on the left side of an equal sign in an assignment statement, or a DO statement, or in a data list in a READ statement or a GET statement. In the definitions below, the item in the data list of a GET or READ statement may be considered to correspond to the item on the left side of the equal sign, in an assignment statement; the value being transmitted may be considered to correspond to the expression on the right side.

COMPLEX (a,b) The letters a and b represent real arithmetic variables that need not have the same characteristics. During execution of an assignment statement, the real part of the expression on the right is assigned to a, the imaginary part to b.

IMAG (c) The letter c represents a complex variable. During execution of an assignment statement, the real value of the expression is assigned to the imaginary part of c.

ONCHAR The expression on the right is converted to a character string of length 1. On assignment, the character that caused the I/O conversion error interrupt

is replaced by the value assigned. This pseudo-variable is defined only while such an interrupt is being processed.

ONFIELD The expression on the right is evaluated and converted to a character string. On assignment, the field that was being processed when the I/O interrupt occurred is replaced by the value assigned. This pseudo-variable is defined only while an interrupt is being processed.

REAL (c) The letter c represents a complex variable. During execution of an assignment statement, the real value of the expression is assigned to the real part of c.

SUBSTR (s,i,k)
(s,i) The letter s represents a string. During execution of an assignment statement, the expression is assigned to the k characters or bits which is the substring of s starting with the ith character or bit. If k is omitted, the expression is assigned from the ith character or bit to the end of the string.

UNSPEC (v) The letter v represents a scalar variable. The expression on the right is evaluated and converted to a bit string and assigned to v without conversion to the type of v.

THE KEY OPTION

Function:

The KEY option is used when direct access to a particular record is required. The file containing this record must be organized in the regional or indexed mode (see "File Organization Attributes," in Chapter 3).

General format:

KEY (expression)

General rules:

1. The "expression," converted to characters, if necessary, is the key value used to locate the particular record in the file. This expression is evaluated whenever transmission of a new record is required.
2. If the KEY option appears in an output context and if the key already exists within the file, that record is replaced by the record being written; if the key does not exist, the record is added to the file.
3. An interrupt on the condition ACCESS (see "ON-Conditions, in Appendix 3) will result if the KEY option is used in an input context and the key does not exist.
4. The KEY option may appear in any of the following statements: GROUP, READ, SEGMENT, SKIP, SPACE, and WRITE.

THE NEWKEY OPTION

Function:

The NEWKEY option serves the same purpose as the KEY option except that the key of the record being written must not already exist within the file. If such a duplicate key exists, an error condition is raised.

General format:

NEWKEY (expression)

General rules:

1. See "The KEY Option" for a discussion of the purpose of the KEY and NEWKEY options.
2. The key of the record is represented by the "expression," which is converted to characters, if necessary. The NEWKEY option can be used in an output context only.
3. The NEWKEY option may appear in either a PUT or a WRITE statement.

THE REGION OPTION

Function:

The REGION option is used when direct access to a file organized in the regional mode is required (see "File Organization Attributes," in Chapter 3, for a definition of the number of records per region in the file).

General format:

REGION (expression)

General rules:

1. The "expression" is converted to integer; this integer <u>n</u> represents the nth subsequent region of the file to which the specified I/O device is to be positioned. The value of the expression must be within the limits of the file.
2. The REGION option may be used in conjunction with the KEY option if the region is defined to contain more than

one record. In this case, the specified device first is positioned to the specified region and the region is then searched for a record with the specified key value; the device is positioned at this record.

3. The REGION option may appear in any of the following statements: GROUP, READ, SEGMENT, SKIP, SPACE, and WRITE.

THE TASK OPTION

Function:

The TASK option specifies that an operation is to be invoked asynchronously and attached as a task.

General format:

```
TASK (task-identifier[,scalar-
   expression])
```

Syntax rule:

The scalar expression that optionally follows the task identifier represents a priority relative to the attaching task.

General rules:

1. The TASK option may be appended to a CALL, FETCH, DISPLAY, OPEN, CLOSE, READ, WRITE, or SORT statement. The appearance of the TASK option in the general format of each of these statements is represented as follows:

   [,task]

   This represents the word TASK, as well as the task identifier and the scalar expression when it appears.

2. A task, represented by the "task identifier," is initiated in one of the following ways:
   a. A procedure is invoked by a CALL statement with a TASK option. The invoked procedure, and all of the procedures it invokes, comprise the attached task.
   b. A DISPLAY, FETCH, SORT, OPEN, CLOSE, READ, or WRITE with a TASK option is executed.

3. The specified task identifier may appear in a WAIT statement or as an argument of the COMPLETE built-in function.

4. When a statement with a TASK option that includes a priority is executed, the scalar expression is evaluated and converted to an integer. If the priority is omitted, it is assumed to be zero, relative to the attaching task.

   If, during program execution, the control system is required to pass control to one of several tasks, the decision is made on the basis of priorities. If the priority relative to the attaching task is zero, as above, a system standard priority is provided.

5. See "Multi-Task Operations," in Chapter 1, for a description of tasks and dynamic program structure.

Examples:

```
1. CALL COMPUTE (ALPHA, BETA,  PHI(3,8)),
   TASK(LWG);
2. DISPLAY    ('TASK'||'LWG'||'ATTACHED'),
   TASK(LWG);
```

ALPHABETIC LIST OF STATEMENTS

The ALLOCATE Statement

Function:

The ALLOCATE statement causes storage to be allocated for specified controlled data.

General format:

See Figure 3 for format of the ALLOCATE statement.

```
                                    ┌initial  [CHARACTER(length)|BIT(length)]    ┐
ALLOCATE identifier [dimension]     │                                            │
                                    └{CHARACTER(length)|BIT(length)}  [initial]  ┘

                             ┌┌initial  [CHARACTER(length)|BIT(length)]    ┐┐
   [,identifier[dimension]   ││                                            ││
                             └└{CHARACTER(length)|BIT(length)}  [initial]  ┘┘...;
```

Figure 3. Format for the ALLOCATE Statement

Syntax rules:

1. Each identifier specifies a scalar, array, or a major structure that is declared with the CONTROLLED attribute.
2. The words "dimension" and "initial" refer to the dimension attribute and the INITIAL attribute.
3. If the dimension attribute is specified (1) it must immediately follow the identifier, and (2) the number of dimensions indicated in the ALLOCATE statement must be the same as the number of dimensions indicated in a DECLARE statement.
4. If the CHARACTER attribute is specified for an identifier in an ALLOCATE statement, the identifier must be declared CHARACTER in a DECLARE statement; if the BIT attribute is specified in an ALLOCATE statement, the identifier must be declared BIT in a DECLARE statement.

General rules:

1. An ALLOCATE statement for an identifier for which storage was allocated and not freed causes storage for the identifier to be effectively pushed down or stacked. This pushing down creates a new generation of data for the identifier. When storage for this identifier is freed, using the FREE statement, storage is effectively popped up or removed from the stack.
2. If different bounds for an array, or length for a string, are specified, the bounds or length given in the ALLOCATE statement override those given in the DECLARE statement.

   In the ALLOCATE statement, the bounds or length may be explicitly stated, the asterisk notation may be used, or there may be no indication of bounds or length.
   a. If explicit bounds or length are given in a DECLARE statement, the ALLOCATE need not specify a dimension or CHARACTER or BIT attribute. If the ALLOCATE statement does not specify bounds or length, the bounds or length are taken from the DECLARE statement. If there is an expression in the DECLARE statement for a bound or the length, it is evaluated and used at the point of allocation. If bounds or length are explicitly specified in the ALLOCATE statement, the specification in the ALLOCATE statement overrides that of the DECLARE statement.

      If asterisks are used in an ALLOCATE statement for bounds or length, then the bounds or length are taken from the most recent generation of data for the identifier. If there is no previous allocation of data, the bounds or length are undefined.
   b. If asterisks are used in a DECLARE statement for the bounds or length of an identifier, then all ALLOCATE statements must have the corresponding dimension and CHARACTER or BIT attributes. The dimension attribute in the logically first ALLOCATE statement for the identifier must have explicit bounds, or the bounds are undefined; the CHARACTER or BIT attributes in the logically first ALLOCATE statement for the identifier must have an explicit length, or the length is undefined. Subsequent ALLOCATE statements may use asterisks for bounds or length.

   Any expressions in the dimension or length specifications in the DECLARE statement are evaluated at the point of allocation.
3. If an INITIAL attribute is given in a DECLARE statement for an identifier that subsequently appears in an ALLOCATE statement, the following additional rules apply:
   a. If the ALLOCATE statement does not specify an INITIAL attribute for the identifier, the initial values are taken from the DECLARE statement. Any expressions in an INITIAL attribute are evaluated at the point of allocation.
   b. If the ALLOCATE statement specifies an INITIAL attribute without the CALL option, initial values are taken from this ALLOCATE statement.
   c. If the ALLOCATE statement specifies an INITIAL CALL attribute, the specified procedure is invoked to perform initialization during execution of this ALLOCATE statement.
4. To determine whether or not storage has been allocated for an identifier the built-in function ALLOCATION may be used.
5. If storage is to be allocated for a major structure, then the entire structure, including all level numbers and identifiers, must be included in the ALLOCATE statement in the same manner as they appear in the DECLARE statement.
6. A parameter that is declared controlled may be specified in an ALLOCATE statement if the associated argument is given the CONTROLLED attribute and is not contained in a structure or is not an element of an array (see

"Relationship of Arguments and Parameters," in Chapter 8).

Examples:

1. The following examples illustrate the use of the ALLOCATE statement when the DECLARE statement contains explicit bounds for an array A:

   DECLARE A(N1,N2) CONTROLLED;
   The values of N1 and N2 are assumed to be known at this point.

N1, N2 = 10;	The bounds are 10 and 10
ALLOCATE A;	
ALLOCATE A (K1,K2);	The bounds are K1 and K2 which override N1 and N2.
N1 = N1 + 1;	
ALLOCATE A;	The bounds are 11 and 10.
ALLOCATE A (*,*);	The bounds are 11 and 10.
ALLOCATE A (J1, J2);	The bounds are J1 and J2.
N1, N2 = 20;	
ALLOCATE A (*,*);	The bounds are J1 and J2.
ALLOCATE A;	The bounds are 20 and 20.

2. The following example illustrates the use of the ALLOCATE statement when the DECLARE statement contains asterisks for the length of a bit string B:

DECLARE B BIT (*)	VARYING CONTROLLED;
ALLOCATE B BIT (*);	Illegal; violates rule 2b.
ALLOCATE B;	Illegal; violates rule 2b.
ALLOCATE B BIT (N);	The length is N.
ALLOCATE B CHARACTER (4);	Illegal; violates syntax rule 3.
ALLOCATE B BIT (8);	The length is 8.

   Further allocations follow the rules illustrated in example (1).

3. The following example illustrates the use of the built-in function ALLOCATION:

```
   .
   .
   .
DECLARE B (N1, N2) CONTROLLED;
IF ¬ ALLOCATION (B) THEN ALLOCATE B;
B = X;
   .
   .
   .
```

4. The following example illustrates the use of the INITIAL attribute for an identifier in an ALLOCATE statement:

```
DECLARE SWITCH BIT(1), A(N,N) CONTROLLED INITIAL ((N*N) 0);
   .
   .
   .
IF SWITCH THEN ALLOCATE A INITIAL (1 (N-1) ((N) 0,1));
   .
   .
   .
ALLOCATE A;
```

The Assignment Statement

Function:

The assignment statement is used to evaluate expressions and to assign values to scalars, arrays, and structures.

General format:

Option 1. (Scalar Assignment)

```
{scalar-variable}  [, scalar-variable]  ...=scalar-expression;
{pseudo-variable}  [, pseudo-variable]
```

Option 2. (Array Assignment)

```
array [,array] ...=array-expression;
```

Option 3. (Structure Assignment)

```
structure-name=structure-expression
   [,BY NAME];
```

Option 4. (Statement Label Assignment)

```
scalar-label-variable
   [,scalar-label-variable] ...=
         {label-constant;       }
         {scalar-label-variable;}

array-label-variable [,array-label-
   variable] ...=
         {label-constant;       }
         {scalar-label-variable;}
         {array-label-variable; }
```

Syntax rules:

1. In Option 1, each variable on the left of the equal sign may be of arithmetic, bit, or character data type.
2. In Option 2, each array referred to on the left of the equal sign may be an array variable name or a pseudo-array.

A pseudo-array is a pseudo-variable with array arguments whose arguments are array variable names. (In the case of the pseudo-variable SUBSTR (s,i,k), this requirement applies only to the argument s; see " Pseudo-Variables.")

All of the arrays on the left and the arrays in the array expression must have the same number of dimensions and identical dimension bounds.

3. In Option 3, in the absence of the BY NAME option, the structure indicated on the left must have identical structuring to the structures indicated in the structure expression.

General rules:

1. The assignment statement is evaluated as follows:
   a. In Options 1 and 4, if any expressions appear on the left of the equal sign, either in subscripts or in pseudo-variables, these expressions are evaluated from left to right. The expression on the right of the equal sign is evaluated. The value of the expression on the right of the equal sign is assigned to the variable on the left.
   b. In Options 2 and 3, the assignment statement is treated as if it were a sequence of scalar assignment statements applied on an element-by-element basis. See Rule 3 below for a discussion of the evaluation of a structure assignment BY NAME.
   c. In the following definition of order of assignment, A is an array of dimensionality n:

```
L1:  DO I1 = LBOUND (A,1)  TO HBOUND (A,1);
     DO I2 = LBOUND (A,2)  TO HBOUND (A,2);
        .
        .
        .
     DO In = LBOUND (A,n)  TO HBOUND (A,n);
     A(I1, I2,...,In) = array-expression;
     END L1;
```

   The result of the evaluation for a later position in an array or structure may be altered by the evaluation and assignment to an earlier position (see Example 1, below).
   d. When necessary, the expression value, or values, is converted to the characteristics of the variable on the left according to the rules in "Expressions," in Chapter 4.
2. When a variable on the left is a bit or character string, the expression is evaluated as above, and the assignment is performed from left to right, starting with the leftmost position.
   a. If the string has a fixed length and the value of the expression is longer than the string, the value is truncated.
   b. If the string has a fixed length and the value of the expression is shorter than the string, the value is extended on the right with zeros for bit strings or with blanks for character strings.
   c. If the string has a varying length and the value of the expression is longer than the maximum length of the string, the value is truncated; the assigned string is of the maximum length.
   d. If the string has a varying length and the value of the expression is shorter than the maximum length of the string, the value is assigned; the new length of the string is the length of the value.
   e. If the variable on the left is the pseudo-variable *SUBSTR with an* argument that is a varying-length string, the assignment is performed to this substring, as follows, with truncation as necessary:
      (1) If the value of the expression is shorter than the substring, the rest of the substring is filled with blanks or zeros precisely as if the name of the specified substring were in an assignment statement.
      (2) If the value of the expression is longer than the substring, it is truncated as if it were in an assignment statement.
      (3) If no substring length is specified, the length of the string becomes i-1 (the number of characters or bits of the string to the left of the ith character or bit) plus the length of the assigned value. Truncation is performed if the new length exceeds the maximum length of the string.
3. If the BY NAME option is used in Option 3, the assignment statement causes the following activity:
   a. Subscript expressions on the left are evaluated.
   b. The names of all contained arrays and scalars are extracted from each structure operand.
   c. These names are qualified by all the minor structure names that contain them, up to but not including the structure names specified in the structure expression BY NAME.

For example, if there are three structures:

```
1 ONE              1 TWO
  2 PART1            2 PART1
    3 RED              3 RED
    3 WHITE            3 GREEN
    3 BLUE             3 WHITE
  2 PART2            2 PART2
    3 GREEN            3 BLUE
    3 YELLOW           3 YELLOW
    3 ORANGE(3)        3ORANGE(3)
  2 PART3
    3 BLACK
    3 WHITE

1 THREE
  3 PART1
    5 BLACK
    5 WHITE
    5 RED
  3 PART2
    5 YELLOW
    5 WHITE
    5 ORANGE(3)
    5 PURPLE
```

Note that the structures contain array names. Also note that the actual level numbers of the two structures do not have to be identical so long as the relative order of the level numbers of the variables is the same.

Consider the statement:

```
ONE = TWO -2 * THREE, BY NAME ;
```

This produces:

```
Structure ONE      Structure TWO
  PART1.RED          PART1.RED
  PART1.WHITE        PART1.GREEN
  PART1.BLUE         PART1.WHITE
  PART2.GREEN        PART2.BLUE
  PART2.YELLOW       PART2.YELLOW
  PART2.ORANGE       PART2.ORANGE
  PART3.BLACK
  PART3.WHITE

Structure THREE
  PART1.BLACK
  PART1.WHITE
  PART1.RED
  PART2.YELLOW
  PART2.WHITE
  PART2.ORANGE
  PART2.PURPLE
```

d. A subset of qualified names is selected. Each selected name is contained in all structures involved in the assignment statement.

From the above example, this produces:

```
PART1.RED
PART1.WHITE
PART2.YELLOW
PART2.ORANGE
```

e. All expressions involving the selected names are evaluated and values from the right are assigned to items on the left for identical qualified names. These assignments must be valid; for example, arrays may not be assigned to arrays of different dimensions or bounds.

For example, the statement ONE=TWO -2 * THREE, BY NAME is then equivalent to:

```
ONE.PART1.RED   =TWO.PART1.RED-2*THREE.
                        PART1.RED

ONE.PART1.WHITE =TWO.PART1.WHITE-2*THREE.
                        PART1.WHITE;

ONE.PART2.YELLOW=TWO.PART2.YELLOW-2*THREE.
                        PART2.YELLOW;

ONE.PART2.ORANGE=PART2.ORANGE-2*THREE.
                        PART2.ORANGE;
```

f. In BY NAME structure assignment, it is unnecessary for the structuring of all participating structures to be identical. Names of variables that are defined on structures appearing in BY NAME assignment take no part in name matching.

4. In Option 4, the value of the label constant or the label variable is qualified by an identification of the current invocation of the block containing the label and by the current task.

   This qualification information is used when a GO TO statement specifies the label variable to make the identified invocation current and to check that control does not cross task boundaries.

Examples:

1. The following example illustrates array assignment (Option 2):

```
Given the array A    2   4
                     3   6
                     1   7
                     4   8

and the array B      1   5
                     7   8
                     3   4
                     6   3
```

Consider the assignment statement:

```
A = (A+B)**2-A(1,1);
```

After execution, A has the value

```
 7  74
93 189
 9 114
93 114
```

Note that the new value for A(1,1), which is 7, is used in evaluating the expression for all other elements.

2. The following example illustrates string assignment:

Given:

A is a fixed-length string whose value is 'XZ/BQ'.
B is a varying-length string of maximum length 8 whose value is 'MAFY'.
C is a fixed-length string of length 3.
D is a varying-length string of maximum length 5.

Then in the statement:

C=A, the value of C is 'XZ/'.
C='X', the value of C is 'Xbb'.
D=B, the value of D is 'MAFY'.
D=SUBSTR(A,2,3)||SUBSTR(A,2,3), the value of D is 'Z/BZ/'.
SUBSTR(A,2,4)=B, the value of A is 'XMAFY'.
SUBSTR(A,4)=B, the value of A is 'XZ/MA'.
SUBSTR(B,2)=SUBSTR(A,2), the value of B is 'MZ/BQ'.
SUBSTR(B,2,2)=SUBSTR(A,3), the value of B is 'M/BY'.
SUBSTR(B,2,2)='R', the value of B is 'MRbY'.
SUBSTR(B,2)='R', the value of B is 'MR'.

3. The following examples illustrate scalar assignment (Option 1):

a. A,B,C = A+SIN(B) + C**2; is the same as X = A+SIN(B) + C**2;
```
   and  A = X;
        B = X;
        C = X;
```

b. COMPLEX (U1, V1) = COMPLEX (U, V) + REAL (Q);

This is the same as:

```
U1=U+REAL(Q);
V1=V;
```

4. The following example illustrates structure assignment (Option 3):

```
DECLARE  1X,  2Y,  2Z, 2R, 3S, 3P, 1A,
    2B, 2C, 2D, 3E, 3Q;
       X = X*A;
```

The second statement is equivalent to the following statements:

```
Y = Y*B;
Z = Z*C;
S = S*E;
P = P*Q;
```

5. The following example illustrates statement label assignment (Option 4);

```
DECLARE P LABEL;
     P = A;
     GO TO P;
     .
     .
     .
     A: X = Y**2;
```

This set of statements causes control to transfer to A when the GO TO P statement is executed.

6. The example below illustrates assignment to an array of structures (Options 2 and 3).

In the following statement, A is an array of structures, and R is a structure:

```
DECLARE 1A(2,2), 2B, 2C, 2D, 3E, 3F,
        1R, 3S, 3T, 3U, 5V, 5W;
```

The following is an array assignment statement:

```
A=R;
```

The above assignment statement is equivalent to the following four structure assignment statements:

```
A(1,1)=R;
A(1,2)=R;
A(2,1)=R;
A(2,2)=R;
```

The four statements above are, in turn, equal to the following:

```
A(1,1)  B, A(1,2).B, A(2,1).B, A(2,2).
   B=S;

A(1,1).C, A(1.2).C, A(2,1).C,  A(2,2).
   C = T;

A(1,1).E,  A(1,2).E, A(2,1),E, A(2,2).
   E = V;

A(1,1).F, A(1,2).F, A(2,1).F, A(2,2).F
   = W;
```

7. The following example illustrates conversion of data defined by a picture

description assigned to floating-point data, and vice versa:

```
DECLARE A FLOAT, B PICTURE '999V99';

A  =  B;   (B is converted from fixed-
           point to floating-point.)

B = A; (A is converted from floating-
           point to fixed-point.)
```

The BEGIN Statement

Function:

The BEGIN statement is the heading statement of a begin block.

General format:

```
BEGIN;
```

General rules:

1. A BEGIN statement is used in conjunction with an END statement.
2. See Chapter 1 for a discussion of blocks and scope of declaration.

Examples:

```
1.  ON OVERFLOW BEGIN;
                    .
                    .
                    .
                  END;

2.  (SIZE): PROCEDURE;
              .
              .
              .
        (NO SIZE): A: BEGIN;
                        .
                        .
                        .
                      END;
              .
              .
              .
           END;
```

The SIZE condition is enabled with the prefix to the PROCEDURE statement. This enabling is negated throughout the begin block with the prefix NO SIZE. On exit from the begin block, SIZE errors are again enabled because statements again are in the scope of the SIZE prefix.

The CALL Statement

Function:

The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of the procedure.

General format:

```
       {entry-name          }
CALL   {(scalar-expression) }

 [(argument [,argument] . . .)] [,task];
```

Syntax rules:

1. The entry name or the value of the scalar expression represents the entry point of the procedure invoked. When necessary, the value of the expression is converted to a character string.
2. Each argument may be any of the following: any type of expression, a statement label constant, a statement label variable, a statement label array, an entry name, an entry parameter, a file name, or a file parameter.

General rules:

1. When the scalar expression is used to designate the entry point and invoked procedure, the scalar expression is evaluated to give a character string. This string specifies a program name that must appear in an active FETCH statement logically prior to the call.

   An active FETCH statement is one whose function has not been voided by a subsequent DELETE statement that specifies the same character value.
2. When the procedure name is represented by a scalar expression, no conversion is performed for the arguments (see "Relationship of Arguments and Parameters," in Chapter 8), and the arguments may not be entry names, statement labels, CONTROLLED variable names, or built-in function names.
3. The TASK option may be specified, when the environment permits, and it is desired that the invoked and invoking procedures be executed asynchronously (see "The TASK Option").
4. See "Relationship of Arguments and Parameters" for a detailed description of the interaction of CALL arguments and invoked entry parameters.

Examples:

```
1.  CALL CRITICAL_PATH (A,B*C,D);
                 .
                 .
```

```
         .
CRITICAL_PATH:   PROCEDURE(ALPHA,BETA,
    GAMMA) FLOAT;
         .
         .
         .
         RETURN;
```

2. `FETCH (A||B);`

```
     .
     .
     .
CALL (A||B) (C,D,E);
```

3. `CALL PAYROLL (NAME, DATE, HRRATE);`

The CLOSE Statement

Function:

The CLOSE statement releases facilities that were allocated during the opening of a file and causes proper disposition of the file.

General format is shown in Figure 4.

Following is the format of "ident":

```
IDENT  {data-list format-list              }
       {entry-name [(argument              }
          [,argument]]...)
```

Syntax rules:

1. The "file name" is the name of a file to be closed, and it may be described in a DECLARE statement with any of the following file attributes: FILE, STANDIN or STANDOUT, MEDIUM, USAGE, POOL, BLOCK and GROUP. These attributes are discussed in Chapter 3.
2. Each file name is separated from its options by a blank.
3. An option that is common to two or more file names may be factored in the same way that attributes in a DECLARE statement may be factored (see "Factoring of Attributes" in Chapter 3).
4. The TASK option may be specified when the environment permits and it is desired to execute the closing activity asynchronously (see "The TASK Option" in this chapter).

General rules:

1. The CLOSE statement causes certain actions to be performed on the file, whose name is one of the file names of the CLOSE statement. The file is repositioned to its logical beginning, and the facilities allocated to it are released.

   If a CLOSE statement is *encountered* and the file has not been opened, the statement is ignored.

   If, however, the file is not closed by a CLOSE statement, the file is closed at the completion of the task.
2. The IDENT option specifying a data list and format list yields a character string that is compared with the file label for an <u>input</u> file or is written as the file label for an <u>output</u> file. The data list and format list are described in Chapter 5.

   The IDENT option specifying an entry name and argument list causes the designated procedure to be invoked for reading or writing the file label.

   If the IDENT option is not specified, no special label operations are performed.

Examples:

1. CLOSE MASTER;

   The file, MASTER, is closed, and the facilities allocated to it are released. The file is repositioned to its logical beginning.

2. CLOSE TABLEA, TABLEB, TABLEC;

   The three files, TABLEA, TABLEB, and TABLEC, are closed in the same way as MASTER, in the preceding example.

```
CLOSE {file-name [ident] ...                                  }   [, task];
      {(file-name [ident] [,file-name [ident] ...)             }
```

Figure 4. General Format for the CLOSE Statement

The DECLARE Statement

See "DECLARE Statements," in Chapter 3, for a discussion of the DECLARE statement.

The DELAY Statement

Function:

The DELAY statement causes execution of the controlling task to be suspended for a specified period of time.

General format:

```
DELAY (scalar-expression);
```

General rule:

Execution of the DELAY statement causes the scalar expression to be evaluated and converted to an integer n and execution to be suspended for n milliseconds.

Execution resumes after n milliseconds only if the controlling task is of sufficiently high priority to be selected in preference to all other ready tasks.

Example:

```
DELAY (10);
```

Execution of the controlling task is suspended for ten milliseconds.

The DELETE Statement

Function:

The DELETE statement causes a specified program to be made inaccessible.

General format:

```
DELETE (scalar-expression);
```

General rules:

1. The scalar expression is evaluated and, where necessary, converted to a character string. This string represents the name of the program to be deleted.

   The DELETE statement makes the specified program inaccessible and also deletes the static data areas of the deleted program.
2. The specified program must have appeared in a previously executed FETCH statement.
3. After execution of a DELETE statement, the program name may not be specified in a CALL statement before it appears in another FETCH statement.

Examples:

1. DELETE('PROCTL');
2. DELETE('A'||'B');

The DISPLAY Statement

Function:

The DISPLAY statement causes a message to be displayed to the machine operator.

General format:

Option 1.

```
DISPLAY (scalar-expression) [,task] ;
```

Option 2.

```
DISPLAY (scalar-expression)
 (character-variable) [,task] ;
```

General rules:

1. Execution of the DISPLAY statement causes the scalar expression to be evaluated and, where necessary, converted to a character string. This character string is the message to be displayed.
2. In Option 2, the character variable represents a string that is a message to be supplied by the operator.
3. The TASK option may be specified, when the environment permits, and it is desired that the DISPLAY statement be executed asynchronously (see the TASK Option).

   In Option 2, when the TASK option is not specified, execution of the program is suspended until the operator's message is received; with the TASK option, execution continues.

   In Option 1, execution continues uninterrupted both with and without the TASK Option.

Example:

```
DISPLAY ('END OF JOB').
```

This statement causes the message, "END OF JOB," to be displayed.

```
Option 1

                              DO;

Option 2.

                 DO WHILE scalar-expression;

Option 3.

                     ⎧pseudo-variable⎫
                 DO  ⎨               ⎬ = specification list;
                     ⎩variable       ⎭

                 A specification list has the following format:

               ⎡TO expression2     [BY expression3]⎤
expression1    ⎣BY expression3     [TO expression2]⎦ [WHILE expression4]

                  ⎡TO expression6     [BY expression7]⎤
[  expression5    ⎣BY expression7     [TO expression6]⎦ [WHILE expression8]]...;
```

Figure 5. General Format for the DO Statement

The DO Statement

Function:

The DO statement delimits the start of a DO group (see "Groups") and may specify iteration of the statements within the group.

General format is shown in Figure 5.

Syntax rules:

1. The "variable" in Option 3 is a subscripted or unsubscripted scalar variable.
2. Each "expression" in the specification list is a scalar expression.
3. If BY expression3 is omitted from the specification list, expression3 is assumed to be one (1).
4. If TO expression2 is omitted from the specification list, the iteration is performed indefinitely unless terminated by some other statement within the scope of the DO.
5. If both TO expression2 and BY expression3 are omitted, this form of the specification list implies a single execution of the DO group with the control variable having the value of expression 1.

General rules:

1. In Option 1, the DO statement delimits the start of a DO group.
2. In Option 2, the DO statement delimits the start of a DO group and specifies an iteration defined by the following expansion:

```
LABEL: DO WHILE expression;
       statement 1
       .
       .
       .
       statement n
       END;
NEXT:  statement
```

The above expansion is exactly equivalent to the following expansion:

```
LABEL: IF ¬ (expression) THEN GO TO NEXT;
       statement 1
       .
       .
       .
       statement n
       GO TO LABEL
NEXT:  statement
```

3. In Option 3 the DO statement delimits the start of a DO group and specifies controlled iteration defined by the following expansion:

```
LABEL: DO variable = expression1
         TO expression2 BY expression3
         WHILE expression4;
       statement 1
       .
       .
       .
       statement n
       END;
NEXT:  statement
```

The above expansion is exactly equivalent to the following expansion:

```
LABEL: variable = expression1;
LABEL1: IF (expression2 -variable) * SIGN
          (expression3) <0
       THEN GO TO NEXT;
       IF ¬ (expression4) THEN GO TO NEXT;
       statement 1
       .
       .
       .
       statement n
       variable = variable + expression3
       GO TO LABEL1;
NEXT:  statement
```

a. If more than one specification is given, the statement labeled NEXT refers to the initialization for the next specification; for example:

```
NEXT: variable = expression5
```

b. If the WHILE clause is omitted, the IF statement involving expression4 is replaced by the null statement.

4. The WHILE clause in Options 2 and 3 specifies that before each associated execution of the DO group, the expression is evaluated and, if necessary, converted to give a bit-string value. If any bit in the resulting string has the value '1', the iterations continue uninterrupted. If all bits have the value '0', the iterations associated with the current specification are terminated.
5. In the specification list, in Option 3, expression1 represents the starting value of the control variable. Expression3 represents the increment to be added to the control variable after each iteration of the statements in the DO group. Expression2 represents the terminating value of the control variable. Iteration terminates as soon as the value of the control variable passes its terminating value. When the last specification is completed, control passes to the statement following the DO group.
6. Control may transfer into a DO group from outside the DO group only if the DO group is delimited by the DO statement in Option 1; that is, iteration is not specified.

Examples:

1. DO INDEX = CTR WHILE A>B, 5 TO 10 WHILE A = B, 100;
2. DO I = J TO K BY I, I+1 TO N BY 1;
3. DO WHILE P;
4. DO;
5. DO WHILE (TAX-DEDCT) < (ESTTAX * 4);

The END Statement

Function:

The END statement terminates blocks and groups.

General format:

```
END [label];
```

General rules:

1. If a label follows END, the END statement terminates that group or block having that label.
2. If a label does not follow END, the END statement terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no other corresponding END statement.
3. An END statement may be used to terminate more than one group or block (see "Use of the END Statement," in Chapter 1).
4. The END statement may have a label preceding the END. This label may be referred to anywhere in the program where the label is known.

For examples, see "Use of the END Statement," in Chapter 1.

The ENTRY Statement

Function:

The ENTRY statement specifies a secondary entry point to a procedure.

General format:

```
entry-name: ... ENTRY [(parameter
    [,parameter]      ...)]
    [data-attributes];
```

General rules:

1. The parameters are names that specify

the parameters of the entry point. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point (see "Relationship of Arguments and Parameters").

2. The data attributes specify the characteristics of the value returned by the procedure when invoked as a function at this entry point. The value specified in the RETURN statement of the invoked entry is converted to the specified data attribute.

   If insufficient data attributes are specified at the entry point, default attributes are applied, as determined by the name of the entry point.

   If an ENTRY statement has more than one label and no data attributes, there is potential ambiguity in the characteristics of the value to be returned, due to default attributes (see "Relationships of DECLARED, IMPLICIT, and Default Attributes"). To avoid this ambiguity, each label is interpreted as if it were a single entry name for a separate ENTRY statement.

Consider the statement:

```
A:I: ENTRY;
```

This statement is equivalent to:

```
A:  ENTRY;
I:  ENTRY;
```

The ENTRY statement must be internal to the procedure block for which it defines a secondary entry point. The ENTRY statement may not be internal to any block contained in this procedure; nor may it be within a DO group that specifies iteration.

Example:

```
CALC4:    PROCEDURE (X,Y,ERROR,Z)
              FLOAT (5);

          ALPHA = SQRT (X**2-Y**Z)
              - ERROR;

          GO TO MEETING;

CALC5:    ENTRY (X,Y,Z) FLOAT (7);

          ALPHA = 2.5E-10 * SQRT
              (X-Y+1);

MEETING:  IF ALPHA>SQRT(ALPHA +2) THEN
              Z = X;

          ELSE Z = Y;

          RETURN (ALPHA * X + Y - Z**-5);

          END CALC4;
```

The EXIT Statement

Function:

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task (see "Multi-Task Operations," in Chapter 1.

General format:

```
EXIT;
```

The FETCH Statement

Function:

The FETCH statement causes a program to be fetched and made available for invocation by a CALL statement, with the entry name specified by an expression.

General format:

```
FETCH (scalar-expression) [,task];
```

General rules:

1. On execution of the FETCH statement, the scalar expression is evaluated and, where necessary, converted to a character string. This string specifies the name of a program to be fetched.

   It is assumed that the specified program was not available before the FETCH.
2. After execution of the FETCH statement, the fetched program may be invoked by a CALL statement, with the entry name specified by an expression (see "The CALL Statement").
3. Data declared external, task identifiers, and file names may be shared only among procedures within a program. Consequently, any program which is made available by a FETCH statement may not share externals with any other program.
4. The TASK option may be specified, when the environment permits, if the FETCH statement is to be invoked asynchronously (see "The TASK Option").
5. Initial values for data in static storage are established at the time of fetching.

Examples:

1. FETCH ('PROCTL');
2. FETCH ('PROG'|| 'BETA'), TASK (RTT) :

```
                    .
                    .
                    .
CALL ('PROG' || 'BETA') ( ALPHA),TASK
 (RTT);
```

The FORMAT Statement

Function:

The FORMAT statement specifies a format list for use with data transmitted under format direction.

General format:

label:...FORMAT format-list;

Syntax rules:

1. The "format list" is as described for use with a format-directed data specification (see "Format Lists" in Chapter 5).
2. At least one "label" is required. It is the name of a statement label appearing in a remote format item.

General rules:

1. A READ, WRITE, GET, or PUT statement may include a remote format specification, R, in the format list of a format-directed data specification. That portion of the format list covered by the R (statement label) format item must be specified in a FORMAT statement with a corresponding statement label.
2. The remote format item and the FORMAT statement must be internal to the same block.

The FREE Statement

Function:

The FREE statement causes the storage most recently allocated for specified variables to be freed. The next most recent allocation is made available, and subsequent references to the identifier refer to that allocation.

General format:

FREE identifier [,identifier] ...;

Syntax rule:

Each identifier is a scalar, array, or structure name of the controlled storage class.

General rules:

1. Controlled storage allocated in a task after it has attached another task cannot be freed by the attached task.
2. If a specified identifier has no allocated storage at the time the FREE statement is executed, no action is taken.

Examples:

1. FREE X,Y,Z;
2. The following excerpt from a procedure illustrates the FREE statement in conjunction with an ALLOCATE statement:

```
DECLARE A(100) INITIAL ((100) 0) CON-
    TROLLED, C(100), X(100);
                  .
                  .
                  .
 ALLOCATE A;
                  .
                  .
                  .
 C=A;
                  .
                  .
                  .
 FREE A;
                  .
                  .
                  .
 X=SIN(C**2 + X/Y);
```

The GET Statement

Function:

The GET statement causes data to be fetched from the current file, converted from external data form, if necessary, and assigned to variables as specified. The GET statement has meaning only in a procedure invoked by a READ or WRITE statement that specifies procedure-directed transmission of data, or in a procedure invoked by such a procedure. See "Procedure-Directed Transmission" in Chapter 5.

General format:

GET data-specification ...;

General rules:

1. The "data specifications" are discussed in Chapter 5. Only those forms

specified for input may be used; a procedure-directed data specification may not be used.

2. As data is fetched from the file, the action that occurs is as if a pointer moved across the records as demanded by the data specifications. This pointer may be repositioned within the record by use of the POSITION statement or the REPOSITION statement.

Example:

```
READAB: PROCEDURE;
        READ (A,B)  (2F(7,3)), CALL GETC;
        .
        .
        .
        END READAB;

GETC:   PROCEDURE;
    GET (C)  (G (8,5));
    .
    .
    .
    END GETC;
```

The GO TO Statement

Function:

The GO TO statement causes control to be transferred to the specified statement.

General format:

$$\text{GO TO}\begin{Bmatrix}\text{label-constant}\\ \text{scalar-label-variable}\end{Bmatrix};$$

General rules:

1. If a label variable is specified, the GO TO statement has the effect of a multi-way switch. The value of the label variable is the label of the statement to which control is transferred.

   Since the label variable may have different values at each execution of the GO TO statement, control may not always pass to the same statement. (Example 2 illustrates a GO TO statement used as a multi-way switch.)

2. A GO TO statement may not pass control to an inactive block (see "Activation and Termination of Blocks," in Chapter 1, for a discussion of active and inactive blocks).

   A GO TO statement may not transfer control from outside a DO group to a statement inside the DO group if the DO group specifies iteration unless the GO TO terminates a procedure invoked from within the DO group.

3. A GO TO statement that transfers control from one block (D) to a dynamically encompassing block (A) has the effect of terminating block D, as well as all other blocks that are dynamically descendant from block A. Conditions are reinstated, and automatic variables are freed in the same way as if the blocks terminated normally. When a GO TO statement transfers control out of a procedure invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued, and control is transferred to the specified statement.

4. A GO TO statement may not terminate a procedure invoked by a statement that specifies the CALL option.

5. A GO TO statement may not be used to transfer control from a task to its attaching task or to any of its ascendant tasks.

Examples:

1.
```
GO TO A234;
   .
   .
   .
A234: ...
```

2. The following example illustrates a GO TO statement that effectively is a multi-way switch.

```
                 .
                 .
                 .
        DECLARE L LABEL  (L1,  L2)  INITIAL
         (L2);
        GO TO MEET;
    L1: X = Y - 1;
        L = L2;
        GO TO MEET;
    L2: Y = X -1;
        L = L1;
  MEET: CALL FUDGE (X, Y, Z);
        IF Z = LIMIT GO TO L;
                 .
                 .
                 .
```

3. The following procedure illustrates use of the GO TO statement with a subscripted label variable to effect a multi-way switch:

```
CALC: PROCEDURE (N1, N2);
      DECLARE SWITCH(3) LABEL INITIAL
       (CALC1, CALC2, CALC3);
      I=MOD(N1+N2,3)+1;
       (GO TO SWITCH (I);
CALC1: ...
      .
      .
      .
      RETURN;
CALC2: ...
```

```
       .
       .
       .
       RETURN;
CALC3: ...
       .
       .
       .
       END CALC;
```

The GROUP Statement

Function:

The GROUP statement causes the group currently being processed to be released from the program.

General format:

```
GROUP [(expression)] [FILE
   (file-name)] [KEY (expression)]
   [REGION (expression)];
```

Syntax rules:

1. Commas must separate the options, which may appear in any order.
2. The "expression," if specified, is a scalar expression and it is evaluated and converted, where necessary, to an integer n. If the expression is not specified, it is assumed to be 1.

General rules:

1. A group is defined as a sequence of records delimited by a group-delimiter. A group is created (1) by a GROUP attribute declared with a file name, (2) by the GROUP format item specified in a WRITE or PUT statement specifying format-directed transmission, or (3) by a GROUP statement.
2. In a GROUP statement, input records are effectively skipped through until a group-delimiter is encountered, with synchronization occurring at the next group, or, if n is the value of the expression, at the nth subsequent group. Output records are terminated with a group-delimiter and released. Empty records are supplied if the data file is defined to have a fixed number of records per group and is an output file.
3. The FILE option specifies that the action is to be taken on the named file. In the absence of a FILE option, the current file is assumed (see "Procedure-Directed Transmission" for a discussion of current files).

   If it is desired to position the standard system input or output tape, the file name in the FILE option must be declared with the STANDIN or STANDOUT attribute, whichever applies.
4. The KEY and REGION options may be used when direct access to a particular record is required (see "The KEY Option" and "The REGION Option," at the beginning of this chapter).

Examples:

1. GROUP FILE (X);

   If X is an input file, records are skipped until a group-delimiter is encountered. The file is then positioned immediately to the right of the group-delimiter.

2. GROUP;

   Since no file is specified, the GROUP statement positions the current file. If the current file is an output file and this current file has not been declared with a GROUP attribute, a group-delimiter is placed on the file, where it is currently positioned, and the group is released from the program.

The IF Statement

Function:

The IF statement causes program flow to depend on the value of an expression.

General format:

```
IF scalar-expression THEN unit-1 [ELSE
   unit-2]
```

Syntax rules:

1. Each "unit" is either a group or a begin block, either of which would be terminated by a semicolon.
2. The IF statement is not itself terminated by a semicolon.

General rules:

1. When the ELSE clause -- ELSE, and its following unit -- is not specified, the scalar expression is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string has the value 1, the unit is executed, and control passes to the statement following the IF statement. If all bits have the value 0, the unit is not executed, and control passes to the next statement. When the ELSE clause is specified, the expression is

similarly evaluated. If any bit is 1, unit-1 is executed, and control passes to the statement following the IF statement. If all bits have the value 0, unit-2 is executed, and control passes to the next statement. The units may contain statements that specify transfer of control (see "Sequence of Control"), and so override these normal sequencing rules.
2. IF statements may be nested, that is, either unit-1 or unit-2, or both, may themselves be IF statements. Since each ELSE clause is always associated with the innermost preceding IF, an ELSE with a null statement may be required to specify the desired effect.

Examples:

1.
```
IF QUEUE = EMPTY THEN CALL COMPILE;
ELSE GO TO MULTIPROCESS;
```

2.
```
A:  IF X > Y THEN
        IF Z = W THEN
        IF W < P THEN Y = 1;
        ELSE P = Q;
        ELSE;
    ELSE X = 4;
J:  Z = 5;
```

The IMPLICIT Statement

See "IMPLICIT Statements," in Chapter 3, for a discussion of the IMPLICIT statement.

The LAYOUT Statement

Function:

The LAYOUT statement specifies the horizontal layout of data on input and output.

General format:

```
LAYOUT
  [FILE (file-name [,file-name] ...)]
  [MARGIN (expression-1, expression-2)]
  [TAB (expression [,expression] ...)];
```

Syntax rules:

1. Commas must separate the options, which may appear in any order.
2. The "expressions" are scalar expressions.

General rules:

1. The FILE option specifies the files to be operated upon. If this option is omitted in a procedure invoked under procedure-directed transmission in a READ or WRITE statement, the files specified in the invoking procedure are used. If the FILE option is omitted elsewhere, the standard output file is assumed.
2. The MARGIN option specifies left and right margins. The values of both expressions are converted to integers when the LAYOUT statement is executed. These values are interpreted as the positions of the left and right margins of the record and line, respectively, relative to the beginning of the record or line. On input, data before the left margin or after the right margin is ignored. On output, the first data item of a record or line is aligned on the left margin, with blanks before it; data is not placed beyond the right margin. Blanks are inserted to the right of the right margin for a fixed-length record.
3. The TAB option specifies tabbing. The expressions are converted to integers when the LAYOUT statement is executed. The values are used to indicate positions from the left end of the line or record. During list-directed and data-directed output, successive items are aligned on successive free tabs. During format-directed transmission, alignment on a tab can be achieved by use of the TAB format item. In other cases, alignment on a tab can be achieved by using the TAB statement.
4. In the absence of a LAYOUT statement, system standards apply.
5. Execution of a LAYOUT statement destroys all options established by a previously executed LAYOUT statement

The Null Statement

Function:

The null statement causes no action and does not modify sequential operation.

General format:

```
[label:] ...;
```

Example:

```
        .
        .
        .
    ON OVERFLOW;
        .
        .
        .
```

The on-unit (see "The ON Statement") is a null statement.

The ON Statement

Function:

The ON statement specifies the action to be taken when an interrupt occurs for the named condition. For a discussion of "enable" and "interrupt," see "Interrupt "Operations," in Chapter 1.

General format:

Option 1

    ON condition [SNAP] on-unit

Option 2

    ON condition SYSTEM;

Syntax rules:

1. The "condition" may be any one of those described in Appendix 3.
2. The "on-unit" is an action specification and it is either an unlabeled single statement or an unlabeled begin block. Since the on-unit itself requires a semi-colon, no semi-colon appears in Option 1.
3. The on-unit may not be a RETURN statement, nor may a RETURN statement be internal to the begin block.

General rules:

1. With two exceptions (see "Programmer-Defined ON-Conditions," in Chapter 1), the standard action to be taken for all ON-conditions is established by the language. When an interrupt takes place before an ON statement for that condition has been executed, standard system action is taken. This standard system action is described in Appendix 3. The ON statement in Option 2 specifies that standard system action is to be taken when an interrupt results from the occurrence of the specified condition.
2. The ON statement in Option 1 is a means for the programmer to specify a special action, that is, execution of the on-unit, to take place when an interrupt occurs for the specified condition.
3. In Option 1, if SNAP is specified, when the given condition occurs, information relevant to the status of the program at the time of the interrupt is listed.
4. Control can reach an on-unit only when an interrupt occurs for the condition associated with this on-unit in an ON statement.
5. If an action specification is established by an ON statement in a given block, it remains in effect throughout this block and throughout all dynamic descendants of this block (see "Activation and Termination of Blocks," in Chapter 1, for a discussion of blocks and generations of blocks).

If an action is specified more than once in a given block, the effect of the old (or prior) ON statement is either temporarily suspended or completely nullified by the new (or later) ON statement, as follows:

a. If the new (or later) ON statement is in a block dynamically descended from the block containing the old (or prior) ON statement, the effect of the old ON statement is temporarily suspended or stacked. The effect of the old ON statement is restored upon termination of the block containing the new ON statement.
b. If the new (or later) ON statement and the old (or prior) ON statement are internal to the same block, the effect of the old ON statement is completely nullified.

6. If an action is specified by an ON statement in a particular task, the effect of this ON statement is inherited by each attached task and by each task attached by the attached task, etc. (see "Multi-Task Operations," in Chapter 1, for a discussion of attached and attaching tasks).
7. The occurrence, during program execution, of an OVERFLOW, UNDERFLOW, ZERODIVIDE, FIXEDOVERFLOW, SIZE, SUBSCRIPTRANGE, CONVERSION, or CHECK (identifier list) condition does not necessarily result in an interrupt even though the action specification is established either by the system or by the previous execution of an ON statement.

For the OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and FIXEDOVERFLOW conditions, an interrupt will <u>not</u> take place on occurrence of the condition if this occurrence is during a calculation lying within the scope of a prefix that specifies NO OVERFLOW, NO UNDERFLOW, NO ZERODIVIDE, NO CONVERSION, or NO FIXEDOVERFLOW (see "Prefixes" and "Scope of Prefixes," in Chapter 1). In any other circumstances, occurrence of one of these five conditions will cause an interrupt, either with standard system action or

with the special action defined by a previously executed ON statement.

For the SIZE, SUBSCRIPTRANGE, and CHECK (identifier list) conditions, an interrupt will take place on occurrence of the condition if this occurrence is during a calculation that is within the scope of a prefix specifying SIZE, CHECK (identifier list) or SUBSCRIPTRANGE. In any other circumstances, occurrence of one of these three conditions will not cause an interrupt.

Thus, the programmer must explicitly enable one of these three conditions using a prefix to specify that he wants an interrupt on the condition. Similarly, he must explicitly designate in a prefix that he wants to suppress the interrupt for OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, or FIXEDOVERFLOW.

Examples:

1.
```
IOPR:  PROCEDURE;
       .
       .
       .
  R1:  READ FILE (FILEX) (A,B)
          (2F(7,3));
       ON CONVERSION (FILEX)
         CONVQ = 9999;
       .
       .
       .
  R2:  READ FILE (FILEX)  (X)
          (A(6));
       END IOPR;
```

Assume that program execution begins with procedure IOPR. At the beginning of execution, all conditions are enabled.

If an illegal character is read from FILEX during the execution of statement R1, the standard system action occurs.

The ON statement specifies that the execution of the statement CONVQ = 9999 is to occur in the event that a conversion error causes an interrupt subsequent to execution of the ON statement. Thus, if a conversion error occurs during the transmission of X in statement R2, the normal sequence of control is interrupted, and the statement CONVQ = 9999 is executed.

2.
```
ZCHK:  PROCEDURE;
       .
       .
       .
  S1:  ON OVERFLOW OVSWCH = 1;
       .
       .
       .
       CALL Q;
       .
       .
       .
   Q:  PROCEDURE;
       .
       .
       .
  S2:  ON OVERFLOW OVSWCH = 2;
       .
       .
       .
  S3:  ON OVERFLOW SYSTEM;
       .
       .
       .
       END Q;
     END ZCHK;
```

Assume that program execution begins with procedure ZCHK.

If an overflow occurs prior to execution of the S1 : ON statement, an interrupt with standard system action occurs. If an overflow occurs subsequent to execution of the S1 : ON statement, an interrupt occurs, and the statement OVSWCH = 1 is executed.

When procedure Q is invoked, the S1 : ON statement remains in effect until the S2 : ON statement is executed. At this point, the effect of the S1 : ON is temporarily suspended, and the S2 : ON goes into effect.

If an overflow occurs between the S2 : ON and the S3 : ON, an interrupt occurs, and the statement OVSWCH = 2 is executed.

When the S3 : ON is executed, it completely replaces the S2 : ON (the S1 : ON is still stacked). If an overflow occurs after the S3 : ON is executed and before the end of procedure Q, it causes the standard system action to take place.

After control is returned from Q to ZCHK, the S3: ON is completely replaced by the S1: ON, whose effect is restored. Any overflows occurring from this point to the end of procedure ZCHK cause the statement OVSWCH = 1 in S1: ON to be executed.

3.
```
SBCHK:  PROCEDURE;
        DECLARE A(9);
B1:   . . .A(I)...;
        ON SUBSCRIPTRANGE BEGIN;
                              IF I>9 THEN
                               GO TO BIGER
```

```
                        ELSE GO TO
                          LITLER;
                        .
                        .
                        .
               BIGER:   ...;
                        .
                        .
                        .
               LITLER: ...;
                        END;
(SUBSCRIPTRANGE) : B2:...A(I...;
        B3:...;
          END SBCHK;
```

Assume that procedure SBCHK is the only procedure in the program.

At the beginning of execution, any occurrence of the condition SUBSCRIPTRANGE will not give an interrupt; it is not enabled, since the condition name does not appear in a prefix in the PROCEDURE statement. However, the occurrence of any other ON condition, except SIZE and CHECK (identifier), will give an interrupt. If in statement B1, the value of I is greater than 9 or less than 1, no interrupt action is taken.

When the ON statement for the condition SUBSCRIPTRANGE is executed, any interrupt that results from a subsequent occurrence of the SUBSCRIPTRANGE condition will result in the action specified by the begin block in the ON statement.

The prefix for statement B2 specifies that the condition SUBSCRIPTRANGE is enabled and should cause an interrupt if it occurs during the execution of statement B2. In this case, the begin block in the ON statement is executed.

In the execution of B3 and subsequent statements, the occurrence of a subscript that is not within the specified range does not cause an interrupt action to occur.

For further examples, see "Interrupt Operations," in Chapter 1.

The OPEN Statement

Function:

The OPEN statement acquires and prepares files for subsequent transmission.

General Format is shown in Figure 6.

"Option" is shown in Figure 7.

Syntax rules:

1. The "file name" may be described in a DECLARE statement with the file attributes FILE, STANDIN or STANDOUT, MEDIUM, USAGE, POOL, BLOCK, and GROUP. These attributes are discussed in Chapter 3.
2. Successive options, which may appear in any order, are separated by blanks, where needed. Each file name is separated from its attributes by a blank.
3. Options that are common to two or more file names may be factored in the same

```
OPEN { file-name [option] ...                                           } [,task];
     { (file-name [option] [,file-name [option]]...)  [option]          }
```

Figure 6. General Format for the OPEN Statement

```
    [ INPUT  ]
    [ OUTPUT ] [TITLE]    [ACTIVITY] [IDENT]  [CREATE]
    [ INOUT  ]
```

Figure 7. Format of "Option" Allowed in the OPEN Statement

way that attributes in a DECLARE statement may be factored (see "Factoring of Attributes" in Chapter 3).

4. The TASK option may be specified when the environment permits and it is desired to execute the opening activity asynchronously (see "The TASK Option").

General rules:

1. The OPEN statement causes certain actions to be performed upon the file, whose name is one of the file names of the OPEN statement. These actions are specified by the options.

   If, however, a file is not opened by an OPEN statement, the file is opened during the first READ or WRITE statement that refers to it.

   If an OPEN statement is encountered for a file already opened, the statement is ignored.

2. The following options may be given to a file in the OPEN statement:

   a. [INPUT | OUTPUT | INOUT]

      One of these options may be given to specify the direction of data transmission that is permitted for the file. INOUT may be given for direct access files, stating that both INPUT and OUTPUT is permitted.

      If one of these options is not specified in the OPEN statement and has not been declared for the file, no assumption is made at the time of execution of the OPEN. If the file is subsequently specified in a READ statement, it is then assumed to be INPUT; if it is specified in a WRITE statement, it is assumed to be OUTPUT. References to the file in PAGE and LAYOUT statements before INPUT or OUTPUT is established, forces no assumptions. References to the file in GROUP, SPACE, SKIP, or SEGMENT statements before INPUT or OUTPUT is established, forces the default assumption INPUT.

      It should be noted that INPUT files may not be written upon and OUTPUT files may not be read.

   b. [TITLE (expression)]

      A file name may be associated with more than one set of data. The choice of the desired set may be delayed until the OPEN statement is executed. At this point, the "expression" in the TITLE option is evaluated, converted to a character string, and used to identify the data set. If the TITLE option is omitted, the file name is taken as the data set name. The TITLE option can be used to let the file name refer to more than one actual file.

   c. [ACTIVITY (expression)]

      The ACTIVITY option causes the "expression" to be evaluated and converted to an integer that indicates the relative activity of the file. This relative activity is represented in units defined individually for each implementation of the language.

   d. [IDENT { data-list format list | entry-name [(argument [,argument]] ...) }]

      The IDENT option in an OPEN statement for an output file specifies that a label is to be placed on the external medium. For an input file, IDENT provides information for label checking.

      The IDENT option specifying a data list and format list yields a character string that is compared with the file label for an input file or is written as the file label for an output file. The data list and format list are described in Chapter 5.

      The IDENT option specifying an entry name and argument list causes the designated procedure to be invoked for reading or writing the file label.

      If the IDENT option is not specified, no special label operations are performed.

   e. The CREATE option specifies that the file is to be created and that any necessary initialization is to be performed. The CREATE option implies the OUTPUT mode.

Examples:

```
OPEN MATRIX INPUT;
```

The file MATRIX is made available for use as an input file.

```
PAGE    [FILE (file-name [,file-name] ...)]
          [NUMBER [expression]] [HEAD (expression)]
          [FOOT (expression)] [SIZE (expression)]
          [SPACE (expression)]
                           ⎧ (expression-2)                                      ⎫ ⎤
        [AT (expression-1) ⎨                                                     ⎬ ⎥
                           ⎩ CALL entry-name (argument [,argument] ...)         ⎭ ⎦ ;
```

Figure 8. General Format for the PAGE Statement

OPEN TABLE OUTPUT;

The file TABLE is made available as an output file and is repositioned at its logical beginning.

The PAGE Statement

Function:

The PAGE statement specifies the pagination of files.

General format is shown in Figure 8.

Syntax rules:

1. Commas must separate the options, which may appear in any order.
2. "Expression" is a scalar expression.

General rules:

1. The PAGE statement causes a skip to the start of the next page or, in a non-print file, a skip to the start of the next group. On output, dummy records or blank lines are generated.
2. The FILE option specifies the files to be operated upon. If the FILE option is omitted and the statement appears in a procedure that is invoked under procedure-directed transmission in a READ or WRITE statement, the files specified in the invoking procedure are used. If this option is omitted elsewhere, the standard output file is assumed.
3. The NUMBER option specifies that the pages or groups are, on input (or are to be on output) numbered on the right of the heading, starting at the number that is the integer value of the expression. If the NUMBER option is not specified, numbering is not expected on input nor is it generated on output. If the expression is omitted, numbering starts at one (1).
4. The HEAD option provides a page title, left adjusted on every page, or the first record of a group for every group. The character string, which is the value of the expression, is the page title. The expression is evaluated when the PAGE statement is executed.
5. The FOOT option provides a left-adjusted line at the foot of each page or the last record of each group. The character string, which is the value of the expression, is the footing line. The expression is evaluated when the PAGE statement is executed.
6. The SIZE option specifies the number of lines per page or records per group, including heading, footing, and blank lines or records. The integer value of the expression provides this information. If this option is unspecified, system standards apply.
7. The SPACE option specifies the line or record spacing. If the integer value of the expression is <u>n</u>, then (n-1) blank lines or empty records are automatically generated on output, or skipped on input, between each line or record explicitly specified. In the absence of this option, SPACE (1) is implied.
8. The AT option specifies that certain action is to occur at a specified location on every page. Expression 1 is evaluated and converted to an integer <u>n</u>. Subsequently, when the nth record of each group, or the nth line of each page is reached, the following occurs:
   a. If expression 2 is specified, it is evaluated and converted to a character string that is understood to appear on input, or is generated as the nth line on output.
   b. If a CALL clause is specified, the arguments are evaluated, where necessary, and the procedure entry, specified by "entry name," is invoked. This procedure may cause special page handling.

   The scope of the arguments is

that of the block containing the PAGE statement. Since the arguments are evaluated at each invocation, the block containing the PAGE statement must still be active at each invocation, that is, when the nth line or record is reached on each page or group.

9. In the absence of a PAGE statement, system standards apply.

Example:

```
PAGE FILE (FILEX), NUMBER (100), HEAD
    ('PAGE HEADING')
FOOT ('BOTTOM OF PAGE'), SIZE (34);
```

The POSITION Statement

Function:

During procedure-directed transmission, the action that occurs is as if a pointer moved across the records as demanded by the data specifications. The POSITION statement manipulates this pointer.

General format:

```
POSITION (format-list);
```

Syntax rules:

1. The format list is as described for format-directed data specification in "Format Lists," in Chapter 5.
2. The following format items are not allowed in the format list of a POSITION statement: GROUP, SEGMENT, SKIP, SPACE, and the remote format item, R.

General rules:

1. When the POSITION statement is executed, the pointer is first reset to the beginning of the current record or segment, if the SEGMENT option has been specified in the READ or WRITE statement that invoked the procedure containing the POSITION statement. The format elements are then used to determine the movement of the pointer as if there were associated data list elements corresponding to the format items. Since no data list exists, all format items must have an explicit or implicit field width (precision) specification.
2. If the POSITION statement moves the pointer across parts of an output record that have no information edited into them, the record is assumed to be initially blank.

Examples:

```
GETAB:  PROCEDURE;

        GET (A,B) (2F (5,2), X (6));

        IF (A>0 & B = 0) THEN POSITION
           (X (25));
        ELSE RETURN;
        RETURN;
        END GETAB;

READY : PROCEDURE;

        READ (Y) (F(8,3)), CALL GETAB;

        END READY;
```

The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

1. Heads a procedure
2. Defines the primary entry point to a procedure
3. Specifies the parameters for the primary entry point
4. Defines any special attributes of the procedure
5. Specifies the attributes of the value that is returned if the procedure is invoked as a function at the primary entry point

General format:

```
entry-name: ...PROCEDURE
        [ (parameter [, parameter]...)]
                               {MAIN      }
        [OPTIONS (max 2 {REENTRANT})]
    [RECURSIVE] [data-attributes];
```

Syntax rules:

1. The "data attributes" are separated by blanks, where necessary.
2. The attributes in the OPTIONS list are separated by commas, where necessary.
3. Commas must separate the procedure attributes (RECURSIVE, OPTIONS), which may appear in any order.

General rules:

1. The "parameters" are names that specify the parameters of the entry point. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point (see "Relationship of Arguments and Parameters," in Chapter 8).

2. The OPTIONS attribute specifies a list of options, separated by commas where necessary. The list, depending upon implementation, includes the options MAIN and REENTRANT. The OPTIONS attribute may be specified only for an external procedure.
3. The RECURSIVE attribute specifies that this procedure may be invoked recursively. It does not apply to contained procedures which, if recursive, must also have the attribute.
4. The data attributes permitted with a PROCEDURE statement are the arithmetic and string attributes. The data attributes specify the characteristics of the value returned by the procedure when invoked as a function at the primary entry point. The value specified in the RETURN statement of the invoked procedure is converted to the specified data attributes.

   If insufficient data attributes are specified at the entry point, default attributes are applied, as determined by the name of the entry point.

   If a procedure has more than one label and no data attributes, there is, because of default attributes, potential ambiguity in the characteristics of the value to be returned (see "Relationships of DECLARED, IMPLICIT, and Default Attributes"). To avoid this ambiguity, the first label is interpreted as if it were a single entry name for a separate PROCEDURE statement, and each subsequent label is interpreted as if it were a separate ENTRY statement.

For example, the statement:

```
A:I:   PROCEDURE;
```

is equivalent to:

```
A:  PROCEDURE;
I:  ENTRY;
```

Examples:

```
B:  PROCEDURE;
       .
       .
       .
    C=A(X,Y);
    END B;
A:  PROCEDURE (B,C) FIXED;
       .
       .
       .
    RETURN (B*C + SIN (P))
    END A;
```

If procedure A is invoked as a function, as in procedure B, then when control is returned to B, the expression (B*C + SIN (P)) is evaluated, converted to fixed point, if necessary, and the value returned to B.

The PUT Statement

Function:

The PUT statement has meaning only in a procedure invoked by a READ or WRITE statement that specifies procedure-directed transmission of data (or in a procedure invoked by such a procedure). The PUT statement then causes data to be fetched from variables as specified and to be moved to the record being constructed for the current file (see "Procedure-Directed Transmission," in Chapter 5).

General format:

```
PUT data-specification ...;
```

General rules:

1. The "data specifications" are discussed in "Data Specifications," in Chapter 5. Only those forms specified for output may be used; a procedure-directed data specification may not be used.
2. As the data record is being formed, the action that occurs is as if a pointer moved across the record as demanded by the data specifications. This pointer may be repositioned by use of the POSITION statement or the REPOSITION statement.

   The character count of varying-length records depends upon the rightmost sweep of the pointer. The character count of fixed-length records is predetermined.

The READ Statement

Function:

The READ statement causes data to be transmitted from an external medium to internal storage.

General format is shown in Figure 9.

Syntax rules:

1. Commas must separate the options, which may appear in any order.
2. Only the "data specification" may appear more than once. However, only

```
READ ⎡ FILE (file-name) ⎤ {data-specification} ...
     ⎣ STRING (name)    ⎦

     ⎡ CROSS [(expression)] [HOLD] ⎤
     ⎣ SEGMENT [(expression)]      ⎦

[PRINT] [COUNT] [KEY (expression)] [REGION (expression)]

[ZERO] [task];
```

Figure 9. General Format for the READ Statement

one procedure-directed data specification can be given, but it may appear in conjunction with other data specifications.

3. When the STRING option is used, only the data specification may be used; the other options must not appear.
4. "Expression" is a scalar expression.

General rules:

1. The programmer may specify a file name by using the FILE option or may specify the name of a string variable or element of a string array by using the STRING option.

   In the absence of a FILE or STRING option, the standard system input file is assumed. The FILE option specifies the name of the file from which the data is to be acquired. It is invalid to specify a file name if that file is in the stack of current files (see "Procedure-Directed Transmission," in Chapter 5, for a discussion of "current" file).

   The STRING option specifies the name of a string from which the data is to be acquired.
2. The data specifications are discussed in "Data Specifications," in Chapter 5. Only those forms specified for input may be used.
3. Each READ statement normally processes one record; an error condition is produced if the data specification causes the record boundary to be crossed. However, the CROSS option permits data acquisition to proceed through any number of records in order to satisfy the specified data requirements. The number of records crossed may be limited by the integer value of the expression in the CROSS option. The margin qualifications for the data file, if specified by a LAYOUT statement, remain valid while under control of the CROSS option. The SPACE, SKIP, and GROUP statements and format items may cause record boundaries to be crossed regardless of the existence of the CROSS option for the file.

   A HOLD option permits part of one record to be processed. This HOLD option causes the position of the record pointer to be set on completion, so that the next READ begins its data scan at the point where the previous operation ceased scanning. If HOLD is not specified, the remaining part of the record is skipped.

   The SEGMENT option implies both the CROSS and HOLD options. The expression in the SEGMENT option is converted, if necessary, to a character string. This string serves as a segment mark. This option permits the data input stream to be synchronized not at the record boundary, but at the mark. Upon satisfying the data requirements for the READ operation, the record is further scanned for the specified mark, so that the next READ of the file proceeds from the mark. All prior unused data is lost to the program. Should the mark be encountered while transmitting data, an end of record condition is raised.
4. The PRINT option specifies that data being read is, at the same time, to be written, in the same format, on the standard output file.
5. The COUNT option specifies that a count of the transmitted scalar data items is to be kept. The COUNT (file-name) built-in function may be used to determine this number of transmitted data items.
6. The KEY and REGION options may be used when direct access to a particular record is required (see "The KEY Option" and "The REGION Option," at the beginning of this chapter).

7. The ZERO option specifies that trailing blanks in numeric data input fields are to be treated as zeros when read under F, G, or E format.
8. The TASK option specifies that the operations associated with the READ statement, including invocation of a procedure specified in a procedure-directed data specification, be performed asynchronously with the procedure containing the READ statement (see "The TASK Option," in this Chapter).

Examples:

1. READ FILE (INVENTORY) , (ITEM.NAME, ITEM.COST) (A(20), F(5,2));

   The file name INVENTORY is read under format-directed transmission for one record. The first 20 characters of the record are placed in the character-string variable ITEM.NAME, the next 5 are converted from fixed-point decimal external format to the internal form of the variable, ITEM.COST. A subsequent READ of the data file is synchronized to the next record boundary.

2. READ FILE (TABLES), (TABLE.POOL) (F(5)), KEY (Q);

   The file named TABLES read for the record is composed of five-digit, fixed-point integers. The record is converted to integer representation, and each item is assigned to the array TABLE.POOL.

3. READ FILE (FILEZ), (AB) (A(10)), SEGMENT ('*');

   The file FILEZ is read for alphabetic data items, each ten characters in length, that are assigned to the character-string array AB. Assignment ceases when either the complete array is satisfied or the SEGMENT mark, the asterisk, is encountered (in the former case, the input data stream is subsequently synchronized to the next occurrence of the segment mark).

4. READ DATA;

   This statement under data-directed transmission specifies that one record is to be read under data-directed transmission from the standard system input tape. It is assumed that the record to be read is composed of scalar assignments giving the name of the data item to be read and the value of this data item.

The REPOSITION Statement

Function:

During procedure-directed transmission, the action that occurs is as if a pointer moved across the records being processed. If an error condition occurs during this activity, then, under the control of an ON statement, a REPOSITION statement may reset the pointer to the position immediately before the data item that caused the error condition.

General format:

```
REPOSITION;
```

Examples:

```
READX:  PROCEDURE;
        READ FILE (FILEX), (X) (F(7,2));
           CALL GETY;
           .
           .
           .
        END READX;

GETY:   PROCEDURE;
        ON CONVERSION (FILEX) REPOSITION;
        GET (Y) (X(17), F(7,2);
        RETURN;
           .
           .
           .
        END GETY;
```

The RESTORE Statement

Function:

The RESTORE statement causes data previously saved by name in auxiliary storage to be restored (see "The SAVE statement").

General format:

```
RESTORE (item, [,item]...)
    [, (expression)] ;
```

Syntax rules:

1. Each "item" may be an unsubscripted scalar, array, or structure name.
2. Each item must have appeared in a previously executed SAVE statement.
3. The "expression" is a scalar expression.

General rules:

1. The RESTORE statement without an expression is equivalent to a series

of simple RESTORE statements as follows:

```
RESTORE (item1);
RESTORE (item2);
.
.
.
```

The RESTORE statement with an expression is equivalent to the following statements:

```
temp=expression
RESTORE (item1),(temp);
RESTORE (item2),(temp);
.
.
.
```

Each simple RESTORE statement causes the specified data to be identified by the value of the "expression," if an expression is specified.

2. Once an item has been restored, it may not be restored again. If the same item has been saved repeatedly with no qualifying expression, the action of restoring the data causes the top item of the stacked information to be deleted. Therefore, the stacked information is treated in first-in last-out manner.
3. An item may be saved in one external procedure and restored in another if the data name is declared EXTERNAL.
4. One SAVE statement may be used to save more than one scalar, array, or structure; however, these items may be restored separately.
5. The extents of the data restored must be the same as the data saved.

Examples:

1. RESTORE (A,B,C);

   Assume that the scalar data items A, B, and C were previously saved by using the SAVE statement. The RESTORE statement then causes A, B, and C to be made available for computation.

2.
```
SAVERM: PROCEDURE
        DECLARE  TABLE(10), 1 RAINBOW,
          2 RED, 3 CRIMSON, 3 PINK,  2
          BLUE,  3  NAVY,  3  TEAL,  2
          YELLOW;
          .
          .
          .
        SAVE (TABLE, RAINBOW);
          .
          .
          .
        RESTORE (TABLE);
          .
          .
          .
        RESTORE (RAINBOW);
          .
          .
          .
        END SAVERM;
```

   Since TABLE is an array and RAINBOW is a structure, the SAVE statement causes all 10 data items in TABLE to be saved and the elementary items (CRIMSON, PINK, NAVY, TEAL, and YELLOW) of the structure to be saved.

   The first RESTORE statement causes the entire array to be restored; the second RESTORE statement causes the elementary items of the structure to be restored.

The RETURN Statement

Function:

The RETURN statement terminates execution of the procedure that contains the RETURN statement and returns control to the invoking procedure. It may also return a value.

General format:

Option 1.

```
RETURN;
```

Option 2.

```
RETURN (expression);
```

General rules:

1. The RETURN statement in Option 1 is used to terminate all procedures except function procedures; control is returned to the first executable statement logically following the statement that invoked the procedure.

   Option 1 represents the only form of the RETURN statement that may be used to terminate a procedure invoked with the TASK option.
2. The RETURN statement in Option 2 must be used to terminate a procedure invoked as a function procedure. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified.

   If the entry point at which the procedure is invoked specifies data attributes, the value of the expression is converted to the implicit or

explicit data attributes specified at the entry point, before it is returned.

Examples:

```
1.    RETURN;

2.    A:  PROCEDURE (X,Y) FIXED;
          DECLARE (X,Y) FLOAT;
          .
          .
          .
          RETURN (X**2+Y**2);
          END;
      B:  PROCEDURE;
          DECLARE A ENTRY FIXED;
          .
          .
          .
          R = A(P,Q);
          .
          .
          .
          END;
```

In the assignment statement (R = A(P,Q);), procedure B invokes procedure A as a function. Procedure B specifies that the scalar expression in the RETURN statement is to be evaluated; since X and Y are floating-point variables and the PROCEDURE statement specifies that the value returned is to be fixed point, the value of the expression is converted to fixed point, and this value is returned to B.

The REVERT Statement

Function:

A REVERT statement specifying a given ON-condition is used to nullify the effect of the most recent previously executed ON statement for that condition and to cause *the action specification* to be reestablished as it was in the immediate, dynamically encompassing block (see "Activation and Termination of Blocks," in Chapter 1).

General format:

REVERT condition;

Syntax rule:

The "condition" is any ON-condition (see Appendix 3).

General rules:

The execution of a given REVERT statement, specifying a given condition and internal to a given block, has the effect described above only if an ON statement, specifying the same condition and internal to the same block, was executed after the block was activated. If such an ON statement was executed, and if the execution of no other similar REVERT statement has intervened, then the execution of the given REVERT statement does have the effect described above. Otherwise, the REVERT statement is effectively treated as a null statement. Thus, a repeated REVERT statement results in no operation.

Examples:

```
  A:  PROCEDURE;
      .
      .
      .
ON1:  ON ZERODIVIDE GO TO ERRSPEC;
      .
      .
      .
      CALL B;
      .
      .
      .
      B:  PROCEDURE;
          .
          .
          .
      ON2:  ON ZERODIVIDE;
            .
            .
            .
            REVERT ZERODIVIDE;
            .
            .
            .
            END B;
      .
      .
      .
ON3:  ON ZERODIVIDE SYSTEM;
      .
      .
      .
      END A;
```

Unless it is stated otherwise, the condition ZERODIVIDE always is enabled. If division by zero occurs prior to execution of statement ON1, an interrupt with standard system action takes place.

If division by zero occurs after execution of ON1 and prior to execution of statement ON2, an interrupt takes place and control transfers to the statement GO TO ERRSPEC.

If division by zero occurs after execution of ON2 and prior to the REVERT statement, an interrupt takes place effectively with no action.

When the REVERT statement is executed, the effect of the statement ON2 is nullified, and statement ON1 again becomes effective. If division by zero occurs after execution of the REVERT statement and prior to the execution of statement ON3, an interrupt takes place, and control transfers to the statement GO TO ERRSPEC.

After the execution of statement ON3, division by zero causes standard system action to take place.

The SAVE Statement

Function:

The SAVE statement causes data to be placed in auxiliary storage, identified by its name.

General format:

```
SAVE(item-1 [,item-2] ...)
    [,(expression)] ;
```

Syntax rule:

Each "item" may be an unsubscripted scalar, an array, or a structure name. The "expression" is a scalar expression.

General rules:

1. The SAVE statement without an expression is equivalent to a series of simple SAVE statements as follows:

   ```
   SAVE (item 1);
   SAVE (item 2);
   .
   .
   .
   ```

   The SAVE statement with an expression is equivalent to the following statement:

   ```
   temp=expression;
   SAVE (item 1), (temp);
   SAVE (item 2), (temp);
   .
   .
   .
   ```

   Each simple SAVE statement causes the specified data to be identified by the "expression" if an expression is specified.
2. If no expression is specified, and items of the same name are repeatedly saved, the values are stacked, and restored in a last-in, first-out basis.

   If an expression is specified and items of the same name are repeatedly stored, only one value for a given name and given expression value is saved at any one time. Subsequent execution of a SAVE statement with matching identification causes the previously saved value to be overriden.
3. The extents of the data, when restored, must be the same as they were when the data was saved.

Example:

```
SAVE (A B, C);
```

The scalar data items A, B, and C are saved in auxiliary storage.

```
.
.
.
DO N=1 TO 10;
X=Y**N;
SAVE (X), (N);
END;
.
.
.
I=J+M/2;
RESTORE (X), (I);
Q = SIN (X);
.
.
```

Each execution of the SAVE statement causes the current value of X to be identified by the current value of N and to be saved.

The RESTORE statement causes one of the previously saved values of X to be restored. In particular, that value of X identified by an integer whose value is J+M/2 is restored.

The assignment statement Q=SIN(X); uses this restored value of X.

The SEGMENT Statement

Function:

The SEGMENT statement specifies positioning within a segmented file. Segment marks may be written on a file by a WRITE statement that specifies a SEGMENT option (see "The WRITE Statement").

General format:

```
SEGMENT [ (expression) ] [ FILE (file-
   name)] [KEY (expression)] [REGION
     (expression)];
```

Syntax rule:

Commas must separate the options, which may appear in any order.

General rules:

1. The scalar "expression," if specified, is converted to character, if necessary; this character string is the segment mark. If the SEGMENT statement appears within a procedure invoked by a CALL option in a READ or WRITE statement that specifies a SEGMENT option, the expression may be omitted; in this case, the SEGMENT option defines the segment mark.
2. The SEGMENT statement causes the pointer to be positioned at the next segment mark after the current position. On input, sufficient records may be skipped to effect this positioning. On output, empty segments may be constructed. Segment positioning need not, however, cross record boundaries.
3. The FILE option specifies that the action is to be taken on the named file. In the absence of a FILE option, the current file is assumed (see "Procedure-Directed Transmission" for a discussion of "current" files).

   If it is desired to position the standard system input or output tape, the file name must be declared with the attribute STANDIN or STANDOUT.
4. The KEY and REGION options may be used when direct access to a particular record is required (see "The KEY Option" and "The REGION Option," at the beginning of this chapter).

Examples:

```
1.  SEGMENT FILE (FILE PROC);
2.  SEGMENT (A+B**3);
3.  DECLARE FILX FILE STANDIN;
    .
    .
    .
    SEGMENT FILE (FILX), KEY (Y);
```

In Example 3, the SEGMENT statement positions the system standard input tape; the name FILX is declared as a pseudonym for this tape.

The SIGNAL Statement

Function:

The SIGNAL statement simulates the occurrence of an ON-condition (see "Interrupt Operations," in Chapter 1, and "The ON Statement"). It may be used to test the condition-recovery routine, the on-unit of the last executed ON statement that has enabled the specified condition.

General format:

SIGNAL condition;

Syntax rule:

The condition may be any one of those described in "ON-Conditions," in Appendix 3.

General rules:

1. When a SIGNAL statement is executed, it is as if the specified condition had actually occurred. The sequence of control through the program is interrupted, and control is transferred to the last executed ON statement that has enabled the specified condition. After execution of the on-unit, control normally returns to the statement immediately following the SIGNAL statement.
2. If an ON statement specifies the CONDITION condition, the condition can cause an interrupt only if a SIGNAL statement, specifying this condition, is given.

Examples:

```
1.     X: PROCEDURE;
          .
          .
          .
     ON1: ON ENDFILE (DATIN) Y,Z = 0;
          .
          .
          .
      S1: SIGNAL ENDFILE (DATIN);
          .
          .
          .
     ON2: ON ENDFILE (DATIN) SYSTEM;
          .
          .
          .
      S2: SIGNAL ENDFILE (DATIN);
          .
          .
          .
          END X;
```

The S1: SIGNAL statement causes an interrupt in the same way as if an attempt to read past a file delimiter, had actually occurred. Control is transferred to the statement Y,Z = 0 in the ON1: ON statement.

When the S2: SIGNAL statement causes an interrupt, control is trans-

ferred to the ON2: ON statement, and standard system action is taken.

2. ON CONDITION (TAX) TAXCT = TAXCT+1;
   .
   .
   .
   SIGNAL CONDITION (TAX);

   The ON statement enables the programmer-specified condition TAX. This condition can occur only when a SIGNAL statement, such as given, causes the condition to occur.

The SKIP Statement

Function:

The SKIP statement causes a specified number of records (or lines) to be skipped.

General format:

```
SKIP [ (expression) ] [ (FILE file-
   name)] [KEY (expression)] [REGION
   (expression)];
```

Syntax rule:

Commas must separate the options, which may appear in any order.

General rules:

1. The scalar "expression," if specified when the SKIP statement is executed, is evaluated and converted, where necessary, to an integer n. If the expression is not specified, n is assumed to be 1.
2. When used with print files, lines and pages are considered, otherwise, records and groups are indicated.

   On input, the SKIP statement causes a skip to the nth record of the group. If the current record is greater than n, a skip to the nth record of the next group occurs.

   On output, the SKIP statement causes the creation of a sufficient number of empty records to cause alignment on the record as described for input.
3. The FILE option specifies that the action is to be taken upon the named file. In the absence of a FILE option, the current file is assumed (see "Procedure-Directed Transmission" for a discussion of current files).

   If it is desired to position the standard system input or output tape, the file name must be declared with the STANDIN or STANDOUT attribute, whichever applies.
4. The KEY and REGION options may be used when direct access to a particular record is required (see "The KEY Option" and "The REGION Option," at the beginning of this chapter).

Examples:

1. OPEN PQR INPUT;

   SKIP (N) FILE (PQR);

   N records are skipped on file PQR.

2. OPEN FILEA OUTPUT;
   .
   .
   .
   SKIP FILEA;

The SORT Statement

Function:

The SORT statement specifies that records on a particular file are to be sorted and, optionally, merged. The sorting is performed on specified fields in ascending or descending order.

General format is shown in Figure 10.

```
SORT FILE (file-name [,file-name]...) [RECORD (format-list)

{ UP   } (integer [,integer]...)...[GIVING (file-name)] [task];
{ DOWN }
```

Figure 10. General Format for the SORT Statement

Syntax rule:

Commas must separate the options and specification, which may appear in any order.

General rules:

1. The size of the records to be sorted either must be specified with a BLOCK attribute for the file name or must be implied by a record description using the RECORD specification.
2. The FILE specification specifies the files to be sorted. If more than one file name is specified, a merge is also performed.
3. The RECORD specification describes either the format of the whole record or merely an initial portion of the record. When only an initial portion of the record is to be described, the BLOCK attribute must be declared for the file name, giving the actual length of the record or the maximum length for varying-length records.

   The format list (see Chapter 5) defines fields on the record; the nth format item describes the nth field. If a format item has an iteration factor of m, this constitutes m fields. Of the positioning format items listed in Chapter 5, only POSITION is permitted; this item does not constitute a field.
4. The UP/DOWN specification indicates the sorting order. UP specifies an ascending sort; DOWN specifies a descending sort.

   The integers in the specification are decimal integer constants that specify the fields to be sorted with respect to the record description. The fields to be sorted are taken from the UP/DOWN specification in left-to-right order. The file is sorted on the leftmost specified field first, then the next field, and so on.

   The sort comparisons are performed using the character collating sequence for character string fields, bit comparison for bit-string fields, and algebraic comparison for arithmetic fields.
5. The GIVING option specifies the file on which the sorted output is to be written. If omitted, the standard output file is used.

   If an output file is specified that differs from the standard output file and that differs from any of the files to be sorted, the file must not be currently open. Rather the operating system opens it for output, produces the sorted file, and closes it.

   If an output file is specified that differs from the standard output file but is identical to one of the files being sorted, then, after reading the file, the operating system closes it, opens it for output, produces the sorted file, and closes it.
6. The TASK option may be specified when the environment permits and it is desired that the SORT statement be executed asynchronously (see "The TASK Option").

Example:

```
SORT FILE(MASTER), RECORD(A(1),
    (6)P'99999'), UP(1,3,5);
```

This SORT statement specifies that the file MASTER is to be sorted in ascending sequence. The RECORD specification indicates that each record in MASTER is composed of 7 fields; the first field contains one character; subsequent fields each contain five digits.

The records are sorted on the first, third, and fifth fields, in that order.

The sorted file is written on the standard output file.

The SPACE Statement

Function:

The SPACE statement causes a specified number of records or lines to be spaced between each record or line explicitly specified.

General format:

```
SPACE [(scalar-expression)] [FILE
   (file-name)] [KEY (expression)]
   [REGION (expression)];
```

Syntax rule:

Commas must separate the options, which may appear in any order.

General rules:

1. The expression, if specified, is evaluated and converted, where necessary, to an integer n. If the expression is not specified, it is assumed to be 1.
2. The FILE option specifies that the action is to be taken on the named file. In the absence of a FILE option, the current file is assumed (see "Procedure-Directed Transmission" for a discussion of current files).

   If it is desired to position the standard system input or output tapes, the file name must be declared with the STANDIN or STANDOUT attribute, whichever applies.
3. The KEY and REGION options may be used when direct access to a particular record is required (see "The KEY Option" and "The REGION Option," at the beginning of this chapter).

The STOP Statement

Function:

The STOP statement causes immediate termination of a program.

General format:

```
STOP;
```

The TAB Statement

Function:

During procedure-directed transmission, the action that occurs is as if a pointer moved across the records being processed. The TAB statement causes this pointer to be aligned on the nth tab of the record or line (see "The LAYOUT Statement"). Intervening data is skipped.

General format:

```
TAB [(scalar-expression)];
```

General rule:

The "scalar expression," if specified, is evaluated and converted to an integer n. This integer represents the nth tab for alignment. If the scalar expression is omitted, TAB (1) is implied.

Example:

```
TAB (X);
```

This statement causes the "pointer" to move to the Xth next tab of the record from its current position, moving to the right.

The WAIT Statement

Function:

The WAIT statement causes program execution to be suspended until certain conditions are satisfied.

General format:

```
WAIT (task-identifier
   [(scalar-expression)]
   [,task-identifier
   [(scalar-expression)]]...)
   [(scalar-expression)];
```

Syntax rules:

1. The "task identifier" must refer to a task that has been attached by the task that executes the WAIT statement.
2. Each "scalar expression" is evaluated and converted to an integer.

General rules:

1. Each attachment of a task with a given identifier causes an entry to be made in an attachment list for that identifier. The first entry in the attachment list indicates the oldest attachment of the task, and the last entry in the list indicates the newest

attachment of the task. A WAIT statement causes interrogation of the status of the attachment list relevant to each specified task identifier.

2. When a scalar expression immediately follows a task identifier, the integer p which represents the value of the expression specifies that only the pth entry in the attachment list for the specified task is to be considered. When the specified task is completed, regardless of whether this pth entry has been interrogated before, this condition is satisfied, and succeeding tasks, specified in the WAIT statement, are interrogated.

   If the attachment list contains fewer than p entries, or if p is zero or negative, the condition is also said to be satisfied.

3. When a scalar expression does not immediately follow a task identifier, the entire attachment list for the specified identifier is scanned from the oldest to the newest entry, as many times as necessary, until one attachment of the specified task is found to have been completed. This entry is then marked so that it will not be interrogated again by other WAIT statements or later in this WAIT statement. When the condition is satisfied, succeeding tasks specified in the WAIT statement are interrogated.

   If the tasks indicated by the attachment list are all complete and are all marked by an earlier interrogation, or if the attachment list is null, the condition is said to be satisfied.

4. When a WAIT statement without the final optional "scalar expression" is encountered, program execution is suspended until each condition is satisfied as described above; control then passes to the statement following the WAIT statement.

   When a WAIT statement with the final optional (scalar expression) is encountered, the expression is evaluated to give an integer m. Program execution is suspended until any m of the n conditions are satisfied. If m is zero or negative, program flow continues. If m is greater than n, all conditions will be satisfied (as described above) before execution continues.

Example:

```
CRANK:  PROCEDURE;
        DECLARE
         (N,M,A(N,M),B(N,M)) AUTOMATIC;
        READ LIST(N,M,A,B), TASK(TAU);
        DECLARE(KAPPA,BETA) EXTERNAL;
        TEMP1 = SIN(KAPPA)**3;
        TEMP3 =  (TEMP1  -  TEMP2)/100
         MOD (MAX(TEMP2, TEMP1), 3);
        WAIT(TAU);
        A = TEMP3*A; B = TEMP3*B;
        WRITE DATA (A,B);
        END CRANK;
```

When the WAIT(TAU) statement is encountered, program execution is suspended. The attachment list for TAU, which contains only one entry for the READ LIST statement, is interrogated. When the execution of the READ statement is completed, execution resumes with the assignment statement following the WAIT.

The WRITE Statement

Function:

The WRITE statement causes data to be transmitted from internal storage to an external medium.

General format is shown in Figure 11.

```
WRITE ┌ FILE (file-name [,file-name] ...) ┐  {data-specification} ...
      └ STRING (name)                     ┘

      ┌ CROSS [(expression)] [HOLD] ┐
      └ SEGMENT (expression)        ┘

       [COUNT] [KEY (expression)] [NEWKEY (expression) [REGION (expression)]

       [FROM (file-name)] [task];
```

Figure 11. General Format for the WRITE Statement

Syntax rules:

1. Commas must separate the options, which may appear in any order.
2. Only the "data specification" may appear more than once. However, only one procedure-directed data specification can be given, but it may appear in conjunction with other data specifications.
3. When the STRING option is used, only the data specification may appear; the other options must not appear.
4. Either a "data specification" or a FROM option must appear.

General rules:

1. The programmer may specify a file name by using the FILE option or may specify the name of a string variable or element of a string array by using STRING option in which case the standard system output file is assumed.

   The FILE option specifies the name of the file to be associated with the data transmitted. A <u>list</u> of file names specifies that these files are to be written simultaneously. It is invalid to specify a file name if that file is in the stack of current files (see "Procedure-Directed Transmission" in Chapter 5, for a discussion of "current" files).

   The STRING option specifies the name of a string into which the data is to be transmitted.
2. The data specifications are discussed in Chapter 5. Only those forms specified for output may be used.
3. Each WRITE statement normally processes one record; an error condition is produced if the data specification causes the record boundary to be crossed. However, the CROSS option permits data transmission to proceed through any number of records in order to satisfy the specified data requirements. The number of records crossed may be limited by the integer value of the expression in the CROSS option. The margin qualifications for the data file, if specified by a LAYOUT statement, remain valid while under control of the CROSS option. The SPACE, SKIP, and GROUP statements and format items may cause record boundaries to be crossed regardless of the existence of a CROSS option for the file.

   A HOLD option permits part of one record to be processed. The HOLD option causes the position of the record pointer to be remembered on completion, so that the next WRITE begins its data scan at the point where the previous one ceased scanning. If a HOLD option is not specified and the record is of fixed length, the remaining part of the record is padded.

   The SEGMENT option implies both the CROSS and HOLD options . The expression in the SEGMENT option is converted, if necessary, to a character string. This string serves as a segment mark. At completion of record construction for the WRITE operation, through one or possibly more records, the specified segment mark is added to the last record. Emission of the record is not thereby implied. Subsequent data is appended to this record until the maximum record length is met.
4. The COUNT option specifies that a count of the transmitted scalar data items is to be kept. The COUNT (file-name) built-in function may be used to determine this number of transmitted data items.
5. The KEY, NEWKEY, and REGION option may be used when direct access to a particular record is required (see "The KEY Option," "The NEWKEY Option," and "The REGION Option," at the beginning of this chapter).
6. The FROM option specifies that the last record read from the file, the name of which is specified in the FROM option, is to be written on the output file. If the last record was read using a KEY option, the key must not be respecified in the WRITE statement since the old key is used.
7. The TASK option specifies that the operations associated with the WRITE statement, including invocation of a procedure specified in a procedure-directed data specification, be performed asynchronously with the procedure containing the WRITE statement (see "The TASK Option").

Examples:

1. WRITE FILE (INVENTORY), (ITEM.NAME, ITEM.COST) (A(20),F(5,2));

   One record of the file named INVENTORY is written under format-directed transmission. The character-string variable ITEM.NAME is written in the first 20 characters of the record. The variable ITEM.COST is converted to fixed decimal and written in the next 5 characters of the record.

2. WRITE FILE (TABLES), (TABLE.POOL) (F(5)), KEY (Q);

   A record is constructed from the contents of the TABLE.POOL array, and

a key, the value of the variable Q, is appended. Should the key be identical to an existent key within the data file, the WRITE causes replacement of that keyed record.

3. WRITE FILE (FILEZ), ('FINAL_DATA',X,Y) (A), (F (3,2), E(5,2));

   Three data items are transmitted to FILEZ (assuming X and Y are scalar variables). The first is the character string of length 10, FINAL_DATA, then the fixed-point form of the value of x, then the floating-point form of the value of Y.

4. WRITE DATA (X,Y,Z);

   The values of the three variables X, Y, and Z are transmitted to the standard output file in the data-directed format. If X is floating-point value 3.1141593, Y is fixed-point value 347, and Z is character-string value bbMATH, then the output data stream would appear as the following:

   X=3.1141593, Y=347, Z='MATH'

   It would be in a form suitable for data-directed input.

A PL/I program may contain compile-time variables and statements. These may be used to effect program parameterization and modification and iterative program generation . The name "macro" is used to describe such compile-time activity. To differentiate the PL/I macro language from the rest of PL/I, the latter is called the "execution-time" language.

When a program involves macro activity, the compiler is supplied with text consisting of a skeleton execution-time program with interspersed macro statements. This text is called the "source text." Compile-time activity is performed by a part of the compiler called the "macro processor." The macro processor converts the source text into text representing an execution-time PL/I program. This created text is called "program text."

For ease of learning, the PL/I macro statements and the variables and labels have been made very similar to the corresponding execution-time entities. With one exception, the macro language forms a subset of the execution-time language. (The exception is the CHARACTER attribute in the DECLARE statement.)

Macro statements and skeleton execution-time text can, in general, be interspersed in any way in the source text. Each macro statement begins with a percent sign. Macro statements are recognized by the occurrence of a percent sign other than within a comment or character string. The restrictions on the form of source text are as follows:

> If the program involves any macro activity, the first statement must be the macro DECLARE statement.
>
> Quotation marks and comments delimiters must be matched within the skeleton execution-time program.

Beyond the requirements above there is no requirement that the skeleton execution-time program be syntactically correct.

MACRO VARIABLES

If compile-time activity is to take place, the source text must contain one and only one macro DECLARE statement. This statement is used to declare the macro variables. All macro variables must be declared. The scope of macro variables is the entire source text. Each macro variable and macro label must have a unique name.

THE MACRO DECLARE STATEMENT

General format:

```
% DECLARE macro-declaration
    [ macro-declaration] ...;
```

A "macro declaration" may be either of the following:

1. macro-variable-specification
2. (macro-variable-specification
   [,macro-variable-specification] ...)
   attribute...

A "macro variable specification" is of form:

```
identifier attribute ...
```

The "identifier" is the name of the macro variable to be declared. Attributes are associated with it in the same way as for the execution time DECLARE statement. In particular, the same rules apply for conflicting attributes. The following are the permissible attributes on a macro DECLARE statement:

FIXED

This specifies that the macro variable is an integer of implementation defined precision.

CHARACTER (decimal-integer-constant)

This specifies that the variable is a fixed-length character string. The length is given by the decimal integer constant.

CHARACTER

If this attribute is supplied, the attribute VARYING must also be given. No length may be specified. These two attributes specify that the macro variable is a varying-length character string. The length of the variable is defined as the length of the last value assigned to it. Initially the length is undefined. (Storage is reserved for the variable as it is needed.)

VARYING

See above.

INITIAL (optionally-signed-decimal-integer-constant)

This attribute may be given only to macro variables with the FIXED attribute. The specified constant is assigned to the macro variable(s) to which the attribute applies. The assignment is performed at the start of compile-time activity.

INITIAL (character-string-constant)

This attribute may be given only to macro variables of character data type. The specified constant is assigned to the macro variable(s) to which the attribute applies. The assignment is performed at the start of compile-time activity.

The first three attributes in the above list are data attributes. One and only one of these must apply to each macro variable.

MACRO EXPRESSIONS

Macro expressions may take one of the following forms:

1. [+ | -] operand [{ + | - | / | * } operand]
2. operand || operand [|| operand] ...

An operand appearing alone in form 1 or the operands in form 2 may be any of the permitted operands. Operands used with a sign or an arithmetic operator must be decimal integer constants or fixed macro variables. Expressions of form 2 may involve conversion from type integer to type character. The conversion is performed according to list-directed transmission rules (see "List-Directed Specifications," in Chapter 5).

Macro expressions are subdivided into fixed and character expressions. Macro expressions involving macro variables that have not been assigned a value are in error.

Macro expressions are evaluated in exactly the same way as execution-time expressions.

EXECUTABLE MACRO STATEMENTS

The executable macro statements are enumerated below. Any executable macro statement may optionally be preceded by a single label, consisting of an identifier.

THE MACRO ASSIGNMENT STATEMENT

General format:

% [label :] macro-variable = macro-expression;

The statement causes the value of the macro expression to be assigned to the macro variable. If the expression is of fixed type, the variable on the left may be either of fixed or character type. If the expression is of character type, the variable on the left must also be of character type unless the value of the expression is a string that contains a decimal integer constant, optionally signed, and with optional surrounding blanks. In this latter case, the variable may be of fixed type.

All conversions implied in assignment are performed according to list-directed transmission rules (see "List-Directed Data Specifications").

THE MACRO NULL STATEMENT

General format:

% label:;

Macro null statements are used for placing macro labels in the text.

THE MACRO GO TO STATEMENT

General format:

% [label:] GO TO label;

The macro label following the GO TO must appear on another macro statement in the source text. The execution of the macro GO TO statement is described below.

THE MACRO IF STATEMENT

General format:

```
% [label]: IF macro-expression
  comparison-operator macro-expression
  THEN GO TO label:
```

The six "comparison operators" are listed in "Scalar Expressions," in Chapter 4. The relationship is evaluated to yield a true value or a false value in the same way as for the execution-time IF statement. If the relationship is true, the statement acts as a macro GO TO statement. If the relationship is false, the statement acts as a null statement.

The two macro expressions to be compared may be of differing data types. In this case, the expression of character type is converted to an integer; it must therefore represent a decimal integer, optionally signed and with optional surrounding blanks.

ACTION OF THE MACRO PROCESSOR

The way in which the macro processor creates program text from source text is described below. Initially program text is null.

Initially the macro DECLARE statement is processed to form a nest of macro variables. INITIAL values are assigned. The DECLARE statement is then deleted from the source text.

After this, the source text is scanned sequentially, starting from the beginning of the text. The scan acts as follows:

If the name of a macro variable occurs in the source text delimited at each end by any of the special characters listed in Chapter 1, other than the break character, and is not within a character string, a comment, or a macro statement, then the scan behaves exactly as if the name had been replaced by the current value of the macro variable it represents, converted to a character string if necessary. The source text, however, remains unchanged. The conceptually inserted value is not enclosed in quote marks. The value must not contain unmatched quotation marks or comment delimiters. If the name of a macro variable occurs within conceptually inserted text, then this name is, in turn, conceptually replaced by its value. (Thus the replacement of JIM by JIM+1 would cause the macro processor to go into an infinite loop.)

If the scan encounters a percent sign, other than in a comment or character constant, a syntactically correct macro statement must follow. This macro statement is executed. If the statement involves a GO TO, the scan resumes at the designated statement. Otherwise, the scan resumes with the text following the statement just executed.

All text passed over by the scan is added to program text with the following provisions:

1. If a macro statement is encountered by the scan, none of the text from the opening percent sign up to, and including, the closing semicolon is added to program text.
2. The circumstances under which redundant blanks or comments appear in program text are individually defined for each implementation of PL/I.

Macro activity ends when the scan encounters the end of the source text. The program text then is compiled normally. It is impossible for the program text to contain any macro statements.

RELATIONSHIP OF ARGUMENTS AND PARAMETERS

When a procedure is invoked, a relationship is established between the arguments of the invoking statement and the parameters of the invoked entry point.

A parameter may be a scalar, array, or structure name (including a label variable name) that is unqualified and unsubscripted, or it may be a file parameter or an entry parameter.

A file parameter may be used within a procedure wherever a file name may be used; an entry parameter may be used wherever an entry name may be used.

A parameter is accessible in the procedure only if the parameter is in the parameter list of the entry point at which the procedure is invoked.

Parameters must be declared in the invoked procedure; they cannot be declared in outer containing blocks. If no explicit declaration is given, an implicit or contextual declaration is assumed, internal to the invoked procedure.

Parameters cannot be declared with the storage class attributes STATIC or AUTOMATIC, with scope attributes, or with the DEFINED attribute.

A parameter may have the CONTROLLED storage class attribute. In this case, the associated argument must also have the CONTROLLED attribute.

EVALUATION OF ARGUMENT SUBSCRIPTS

When an argument is a subscripted variable, the subscripts are evaluated before invocation. The specified element is then passed as the argument. Subsequent changes in the subscript during the execution of the invoked procedure have no effect upon the corresponding parameter.

USE OF DUMMY ARGUMENTS

There are a number of circumstances under which an argument, as it appears in a procedure reference, will <u>not</u> be passed. In this case, any changes to the corresponding parameter in the invoked procedure will <u>not</u> be reflected as changes in the value of the original argument.

If an argument is an expression in which an operator or constant appears anywhere other than in a subscript, then a dummy argument is passed to the invoked entry.

Other cases in which a dummy argument is constructed are described below.

USE OF THE ENTRY ATTRIBUTE

An ENTRY attribute may be specified for the invoked entry name; this ENTRY attribute appears in a DECLARE statement in the invoking procedure. If an ENTRY attribute is not specified in the invoking procedure for the invoked entry name, the attributes of the arguments must agree with those of the corresponding parameters of the invoked entry.

If an ENTRY attribute without parameter attribute lists is specified for an identifier, it indicates that the identifier is an entry name. In this case also, the argument and parameter attributes are assumed to agree.

However, if an ENTRY attribute with parameter attribute lists is specified for the invoked entry name, then the attributes of the parameter of the invoked entry are assumed to be the same as those specified for it in the ENTRY attribute specification. If an argument has data attributes that differ from the corresponding set of attributes defined in the ENTRY attribute specification, then a dummy argument, with the value of the given argument, is constructed by converting the argument to the data attributes defined for the corresponding parameter in the ENTRY attribute specification. If conversion is impossible, then the program is in error (e.g., conversion of file name to bit). The dummy argument is then passed to the invoked entry.

Example:

```
A:  PROCEDURE;
    DECLARE B ENTRY (FIXED, FLOAT),
        (C,D) FLOAT;
         .
         .
```

```
         .
     CALL B(C,D);
         .
         .
         .
     END A;

B:   PROCEDURE (P,Q);
     DECLARE P FIXED, Q FLOAT;
         .
         .
         .
     END B;
```

The specification of the ENTRY attribute in procedure A indicates that B has two parameters, the first with attribute FIXED and the second with attribute FLOAT. However, the arguments C and D both have the FLOAT attribute. Since C is to be fixed point when it is passed to procedure B, a dummy argument is constructed by converting C from floating point to fixed point. This dummy argument is then passed to B.

CORRESPONDENCE OF PARAMETERS AND ARGUMENTS

If a parameter of an invoked entry is a scalar, the argument must be a scalar expression. The data attributes of the argument must agree with the corresponding attributes of the parameter.

If a parameter of an invoked entry is an array, the argument must be an array expression. The argument may also be a scalar expression so long as an ENTRY attribute is given for the invoked entry, specifying the dimension attribute for the relevant parameter. In this case, a dummy array argument will be constructed where the value of each element of the array is the value of the scalar expression. The data attributes of the argument must agree with those of the parameter. If the asterisk notation is not used to specify the dimensions of the parameter in the invoked procedure, the values of the bounds of the array argument must agree with the values of the bounds specified for the parameter in the invoked procedure.

If a parameter is a structure, the argument must be a structure expression. When a structure description is given for a parameter in an ENTRY attribute specification, a scalar expression may be specified as the corresponding argument. A dummy structure argument will then be constructed where the value of each element of the structure is the value of the scalar expression. The data attributes of the elements of the structure argument must match those of the associated parameter as specified in the invoked procedure. The relative structuring of the argument and the parameter must be the same, although the level numbers need not be identical.

If a parameter is a scalar-label variable, the argument must be a scalar-label variable or constant. If a parameter is an array-label variable, the argument must be an array-label variable. If an ENTRY attribute is given for the invoked entry in the invoking procedure, and if the appropriate parameter attribute list specifies that the parameter is a label array, then the argument may be a scalar-label variable or constant; a dummy label array argument will be suitably constructed. A dummy argument is always constructed when the argument is a label constant or label array.

If the argument is a statement label constant, this statement label constant is qualified by an identification of the current invocation of the block containing the label and by the current task; this information is passed as a dummy argument to the invoked entry.

If a parameter is an entry parameter, the argument must be an entry name or entry parameter. When a parameter is specified as an entry parameter in the parameter description of an ENTRY attribute and is not given data attributes, no default data attributes are assumed. If it is necessary that the entry parameter have data attributes, they must also be specified in the parameter description.

If a parameter is a file parameter, the argument must be a file name or file parameter.

An argument passed to a parameter that is a fixed-length string variable must be a fixed-length string. An argument passed to a parameter that is a varying-length string variable must be a varying-length string.

Example:

```
M1: PROCEDURE;
    DECLARE A(10), AA(10), AAA(10),
       N EXTERNAL;
          .
          .
          .
    N=10; CALL S1(A,AA,AAA);
          .
          .
          .
    END M1;
```

STACKING OF TASK IDENTIFIERS

When a task is attached, the task identifier is recorded in a list of active tasks within the attaching task. This list contains the identifiers of all active tasks that have been attached by the task but have not yet been recognized as complete by a WAIT statement. When a task is recursively attached, the list element is a stack which operates on a last-in first-out basis.

The list of task identifiers is only known to the task that generated the list. The list is not copied into the storage of attached tasks that initially have null lists. Thus, there can be no reference by one task to a task that was attached by some other task. A task cannot wait on, or test with the COMPLETE built-in function, any task other than its own immediate descendants.

ABNORMALITY

The ABNORMAL, NORMAL, USES, and SETS attributes are provided in PL/I to enable the compiler to generate optimized code. Although the actual degree of optimization will vary among compilers, these attributes are provided to indicate situations where it is not possible to exploit the appearance of common expressions that are in a statement or among a group of statements.

In the absence of any information, the following assumptions are made:

1. All external function references are normal.
2. All other procedure references are abnormal.
3. All variables are normal.

A variable is said to be <u>abnormal</u> if its value may be altered without an explicit indication. Thus, for example, the appearance of a variable name on the left side of an assignment statement, in the data list specification of a READ or GET statement, or as an argument to an abnormal function or procedure (see below) indicates a predictable situation where the variable may change its value. However, when the variable is subject to change by the occurrence of an ON-condition, or if it is subject to change in a procedure invoked with the TASK option (see "Multi-Task Operations"), then there is no way to predict the point at which the change in value will occur or, in fact, if it will occur.

Such possibilities cannot always be recognized contextually. Furthermore, if a portion of a source program contains several references to such a variable, the order in which the indicated operations are executed becomes significant. (For example, if B is abnormal, the expression B + B is not necessarily equivalent to the expression 2 * B.)

The implication is that the programmer expects the operation to be performed in a particular order. Such variables must therefore be declared ABNORMAL, to inhibit the optimization of such portions of a source program.

A procedure may possess varying degrees of abnormality. A procedure is said to be "definitively abnormal" if it, or any procedures invoked by it, accesses, modifies, allocates, or frees external data or modifies, allocates, or frees arguments. In addition, an internal procedure is abnormal if it, or any procedures invoked by it, accesses, modifies, allocates, or frees any variables known in the invoking block. Such procedures are only definitively abnormal because the exact nature of their abnormality is described by the USES and SETS attributes, thus inhibiting some, but not all, optimization in the neighborhood of a reference to the procedure (see "The USES and SETS Attributes" in Chapter 3).

However, if a procedure is "completely abnormal," all optimization of successive references must be inhibited. A procedure is completely abnormal if it, or any procedures invoked by it, does any of the following:

1. returns inconsistent function values for identical argument values
2. maintains any kind of a history
3. performs input or output operations
4. returns control from the procedure by means of a GO TO statement

The ABNORMAL attribute (described in Chapter 3) is used to describe such a procedure. It may also, of course, be used to describe a procedure that is "definitively abnormal."

When abnormality is specified, the order of execution becomes significant. In particular, if an expression contains a reference to an abnormal function that may affect values in other parts of the expression, the value of the expression will, in general, depend upon the order in which data is accessed (see "Order of Evaluation of Expressions," in Chapter 4).

ARITHMETIC GENERIC FUNCTIONS

The generic functions listed in this section return a value of type arithmetic. The arguments may, unless otherwise specified, be any expressions. If nonarithmetic they will be converted to type arithmetic before the function is invoked according to the rules stated under "Scalar Expressions". Where reference is made to an argument, it should be taken to mean the converted argument when a nonarithmetic argument has been specified. The magnitude of a complex number is the positive square root of the sum of the squares of the real and imaginary parts.

Name	Arguments and Function Value
ABS	Arguments: One is permitted. Function value = absolute value of argument, i.e., positive value of real argument, positive magnitude of complex. Base, scale, and precision are those of argument.
MAX	Arguments: Two or more are permitted. Function value = value of maximum argument, converted to highest characteristics of all arguments specified. The magnitude of complex numbers is used for comparison.
MIN	Arguments: Two or more are permitted. Function value = value of minimum argument, converted to highest characteristics of all arguments specified. The magnitude of complex numbers is used for comparison.
MOD	Arguments: Two are permitted, x and y. Function value = x-FLOOR (x/y) *y. The rules of expression evaluation give characteristics of result. If the value obtained by this formula is negative, the absolute value of the modulus y is added to give a positive result.
SIGN	Arguments: One is permitted. Function value = integer 1 if argument >0; = 0 if argument = 0; = -1 if argument <0.
FIXED	Arguments: Three are permitted. The second and third are optional decimal integers specifying the number of digits after the decimal or binary point. If omitted, the second argument assumes a value specified by each implementation, the third assumes zero. Function value = first argument converted to fixed-point scale with precision as specified but base and mode unchanged.
FLOAT	Arguments: Two are permitted. The second is an optional decimal integer specifying the precision of the result. If omitted, a value specified by each implementation will be assumed. Function value = first argument converted to floating-point scale with precision as specified but base and mode unchanged.
FLOOR	Arguments: One is permitted, x. If the argument is complex, it will be converted to real. Function value = largest integer not exceeding x. Base, scale, mode, and precision are that of converted argument.
CEIL	Arguments: One is permitted, x. If the argument is complex, it will be converted to real. Function value = smallest integer not exceeded by x. Base, scale, mode, and precision are that of converted argument.
TRUNC	Arguments: One is permitted, x. If the argument is complex, it will be converted to real. Function value = FLOOR (x) if x ≥ 0, = CEIL (x) if x < 0. Base, scale, mode, and precision are that of converted argument.
BINARY	Arguments: Three are permitted. The second and third are optional decimal integers specifying the binary precision of the result.

implementation defined precision giving current length of argument.

HIGH

Arguments: One is permitted, a decimal integer constant.

Function value = character string of the length specified and composed of the highest characters of the data character set.

LOW

Arguments: One is permitted, a decimal integer constant.

Function value = character string of the length specified and composed of the lowest characters of the data character set.

REPEAT

Arguments: Two are permitted. The first is a string and the second a decimal integer constant n.

Function value = string argument concatenated with itself n-1 times.

UNSPEC

Arguments: One is permitted.

Function value = bit string which is the internal coded representation of the argument. The length is an implementation defined function of the argument characteristics.

BF$n^1n^2n^3n^4$ where each term n is either 0 or 1.

Arguments: Two are permitted, bit strings X and Y.

Function value = bit string Z where if X and Y are of different lengths, the shorter is extended with zeros, and Z is of the longer length. The following table relates the jth bit of Z to the jth bits of X and Y.

Xj	Yj	Zj
0	0	n^1
0	1	n^2
1	0	n^3
1	1	n^4

BUILT-IN FUNCTIONS FOR MANIPULATION OF ARRAYS

The following built-in functions have array expression arguments and return scalar values. In the following functions X is any array expression unless otherwise specified.

Function Reference	Function Value
SUM (X)	A scalar value equal to the sum of all the elements of X. Precision, scale, mode and base are that of argument elements. (The argument is converted to arithmetic before the function is invoked.)
PROD (X)	As above but product.
ALL (X)	The argument is converted to bit string. The result is a bit string of the length (or max length if variable) of the elements of X. The ith bit of the results is 1 if the ith bits of all the elements of X are 1. Otherwise 0.
ANY (X)	As above, ith bit of the result is 1 if any of the ith bits of the elements of X are 1. If all 0, then the result bit is 0.
POLY (X,Y)	X (M:N) and Y (P:Q) are vectors. Result is: $\sum_{J=0}^{N-M} (X(M+J) * \prod_{I=J}^{N-M} Y(P+I))$ If (Q-P) < (N-M), then Y(I)=Y(Q) when P+I>Q. This definition permits a scalar as second argument, when both P and Q are taken as 1. The characteristics of the result are the higher of the arguments after conversion to arithmetic.
LBOUND (X,S)	S is a scalar expression which is converted to an integer n. The function value is an integer giving the current lower bound of the nth dimension of X.
HBOUND (X,S)	As above but higher bound.
DIM (X,S)	S is as above. The function value is an integer giving the current extent of the nth dimension of X.
SCAN (A,I, 'operator')	A is any array expression; I is a decimal integer constant. The third argument may be any operator in quotes. The function value is defined by the

value of TEMP on exit from the following loop:

```
TEMP = A (*,.....,*, LBOUND (A,I) ,
  *,.....,*) ;
DO J = LBOUND (A,I) + 1 TO HBOUND
  (A,I) ;
TEMP = TEMP operator A (*,.....,*,J,
  *,.....,*) ;
END;
```

TEMP has dimensions N-1 where A has N. The bounds of TEMP are the first (I - 1) and the last (N - I) of A. TEMP has the base, scale, mode and precision of A if arithmetic, and the length of elements of A, if string.

ARRAY BUILT-IN FUNCTIONS

All the built-in functions listed under "Arithmetic Generic Functions" and "String Generic Functions" in this appendix may have array expressions as main argument. They yield an array of the same dimensions and bounds as the argument, the function being performed on each element. The rules are the same as those for the scalar functions.

CONDITION BUILT-IN FUNCTIONS

The following built-in functions (with no arguments) are available to allow investigation of interrupts arising from enabled ON conditions. They may be referred to only in ON units.

Function Reference	Function Value
ONPOINT	An integer, being the value of the I/O buffer pointer when the I/O condition arose.
ONLOC	A character string of variable length, being the name of the procedure in which the condition arose.
ONFIELD	A character string of variable length, being the contents of the field being processed when the I/O condition arose.
ONCHAR	A character string of length 1, being the character which caused an I/O conversion error.
ONCODE	An integer whose value depends on a detected error. Each of the following error categories has a set of contiguous code values: I/O errors Conversion errors Control program errors

OTHER BUILT-IN FUNCTIONS

Function Reference	Function Value
DATE	Character string of length six of the form YYMMDD, where YY is year, MM is month, DD is day.
TIME	Character string of length nine of the form HHMMSSTTT, where HH is hours, MM is minutes, SS is seconds, TTT is milliseconds.
ALLOCATION (X)	X is a CONTROLLED major structure or unsubscripted array or scalar variable not in a structure. The function value is '1'B if storage has been allocated for X and '0'B if not.
POINT (Filename)	The value of this function is a decimal integer precision (Y), where Y is implementation defined. It specifies the current position of the pointer relative to the start of the current logical record for the named file.
COUNT (Filename)	The value of this function is a binary fixed-point integer of default precision. It returns a value that is the number of scalar data items transmitted during the last read or write operation on the specified file, which contained the COUNT option.
COMPLETE (Task Identifier)	The task identifier refers to the task most recently attached by the task containing the COMPLETE function reference, but not waited on. If the task specified has been completed, or is unknown (i.e., because it has never been attached, or has already been successfully waited on) the function value is '1'B. Otherwise it is '0'B.

ROUND (Expression,
Decimal Integer Constant)

The expression may be scalar array, or structure. The function value is the expression value rounded on the n'th digit after the point where n is the value of the integer. (Binary digits if binary base, decimal if decimal base.) Nonarithmetic elements of the expression argument are unmodified. Floating point rounding is a bias removal rather than systematic rounding. Base, scale, mode and precision of the value is that of argument. For fixed-point scale, digits after the rounded digit are set to zero.

STRING (Structure Name)

The argument must be a packed structure composed either of all bit strings and numeric fields of binary base, or character strings and numeric field of decimal base. The function value is a string, being the concatenation of all the structure elements.

DIGIT POINT AND SUBFIELD DELIMITING CHARACTERS

9 Specifies that the associated field position will contain any decimal digit.

1 Specifies that the associated field position contains a binary digit. This character may not appear in a picture with either 2 or 3.

2 Specifies that the associated field position contains a binary digit, being part of a binary value in 2's complement notation. This character may not appear in a picture with either 1, 3, or S.

3 Specifies that the associated field position contains a binary digit, being part of a binary value in 1's complement notation. This character may not appear in a picture with either 1, 2, or S.

V Specifies that a decimal or binary point should be assumed to appear at this point in the associated field. It does not specify a character in the field.

K Specifies that the exponent subfield should be assumed to follow the point in the field associated with the K. It does not specify a character in the field.

E Specifies that the associated field position will contain the letter E, indicating the start of the exponent subfield.

ZERO SUPPRESSION CHARACTERS

A leading zero in a numeric subfield is a zero to the left of the actual occurrence of the digits 1 to 9 in the subfield. The leftmost of these latter digits and all digits in the subfield following it, are significant digits (including any zeros). Picture characters are provided for zero suppression, leading zero suppression, and the replacement of these zeros by blanks or asterisks.

Z Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by a blank, otherwise the digit will appear. The character may not appear to the right of 9 T I R or a drifting string in a subfield. It may not appear with * in a subfield.

* Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by *, otherwise the digits will appear. The character may not appear to the right of 9 T I R or drifting string in a subfield. It may not appear with Z in a subfield.

Y Specifies a conditional digit position. If the associated field position involves a zero (leading or otherwise) it will be represented in the field by a blank, if it involves a digit other than zero that digit will appear.

DRIFTING EDITING SYMBOLS

The following picture characters may be static or drifting:

Character	Name
S	
+	sign characters
-	
$	dollar sign

The static use of these characters specifies that there is a field position where a sign, a dollar sign, or a blank <u>always</u> appears. The drifting use specifies that leading zeros may be suppressed, and the suppressed positions may contain blanks. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a dollar sign.

A drifting character is specified by multiple use of that character in a picture subfield. Thus, if a subfield contains one dollar sign, it is interpreted as static; if it contains more than one, as drifting. The drifting character must be specified in each position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing interspersed editing characters comma (,), point (.), slash (/), or V or B. Picture characters slash, comma, point, and B following the last drifting symbol of the string are considered part of the string. However, a following V terminates the string and is not part of it. A subfield

may only contain one drifting string. The picture characters * and Z may not appear to the right of a drifting string in a subfield.

The field position associated with the character slash, comma, point, and B appearing in a drifting string will contain one of the following:

1. slash, comma, point, or blank if a significant digit has appeared to the left
2. the drifting symbol, if the next position to the right contains the leftmost significant digit of the subfield
3. blank, if the leftmost significant digit of the subfield is more than one position to the right

If a drifting string contains the drifting character n times, then the string is associated with n - 1 conditional digit positions. The field position associated with the leftmost drifting character may only contain the drifting character or blank, never a digit. If a drifting string is specified for a subfield, the other potentially drifting characters may only appear once to the left of the string in the subfield, i.e., the other characters represent a static sign or dollar sign.

If a drifting string contains a V, then all digit positions of the subfield following the V must also be part of the drifting string.

If one of the characters Z or * follows the V in a subfield, then all digit positions in the subfield following the point must be Z or asterisk (*).

In the case where all digit positions after the V contain suppression characters, suppression will only occur where all the fraction digits are zero. The resulting field will then be all blanks or asterisks. If there are any significant fraction digits they all will appear unsuppressed.

DRIFTING CHARACTERS

$ If this character appears more than once in a subfield it is a drifting character, otherwise it is a static character. The static character specifies that the character $ be placed in the associated field position. The static character must appear either to the left of all digit positions in a subfield or to the right of all digit positions in a subfield. See details above for the drifting use of the character.

S Specifies the sign character + if the field value is ≥0, otherwise -. The character may be drifting or static. The rules are identical to those for the dollar sign.

\+ Specifies the sign character + if the field value is ≥ to 0, otherwise blank. The character may be drifting or static. The rules are identical to those for the dollar sign.

\- Specifies the sign character - if field value is <0, otherwise blank. The character may be drifting or static. The rules are identical to those for the dollar sign.

EDITING CHARACTER

B Specifies that a blank appear in the associated field position.

CONDITIONAL EDITING CHARACTERS

, If the subfields in which the comma appears involve no zero suppression, that character specifies that a comma will appear in the associated field position. If zero suppression is involved the comma will appear only if there is an unsuppressed digit to the left of the comma position in the subfield. If there is no such unsuppressed digit, the associated field position will contain a character that depends on the first digit (conditional or otherwise) picture character preceding the comma.

If the preceding character is an asterisk the field position will contain an asterisk.

If the preceding character is a drifting sign or dollar sign the action taken will be identical to that which would have occurred if the picture specification had contained the drifting character in place of the comma.

If the preceding picture character is anything other than the above, the field position associated with the comma will contain a blank.

/ Exactly as comma, but a slash will appear when indicated.

. Exactly as comma, but a point will appear when indicated.

SIGN CHARACTERS

Digit characters in numeric fields may contain an overpunched sign. The following picture characters are used to specify overpunching:

T Specifies that the associated field position will contain a digit overpunched with the sign of the containing subfield.

I Specifies that the associated field position will contain a digit overpunched with + if the containing subfield is ≥ 0; otherwise it will contain the digit with no overpunching.

R Specifies that the associated field position will contain a digit overpunched with - if the containing subfield is < 0; otherwise it will contain the digit with no overpunching.

The two character picture items CR and DB may be used to reflect the sign of numeric fields.

CR Specifies that the associated field positions will contain the letters CR if the containing field value is <0. Otherwise the positions will contain two blanks. The characters CR may appear only to the right of all digit positions of a field.

DB As CR, except that a DB appears if the containing field value is greater, or equal to, zero.

SCALING FACTOR CHARACTER

F Specifies the location of the decimal or binary point in a fixed-point number.

STERLING PICTURES

The following additional characters are provided for use in sterling pictures.

8 Specifies the position of a shilling digit in BSI single-character representation.

7 Specifies the position of a pence digit in BSI single-character representation.

6 Specifies the position of a pence digit in IBM single-character representation.

G Specifies the start of a sterling picture. It does not specify a character in the numeric field.

H Specifies that the associated field position contains the shilling character S.

D Specifies that the associated field position contains the pence character D.

PICTURES FOR CHARACTER STRINGS

A form of picture may be given for character strings. The following are used to indicate the form:

A The associated field position may contain any letter or blank.

X The associated field position may contain any character.

Editing characters are provided.

The ON-conditions are those conditions that may be specified in the ON statement. These conditions are also specified in SIGNAL and REVERT statements.

For each condition name, the description in this appendix includes the circumstances under which the condition occurs, the standard system action that would be taken in the absence of programmer-specified action, and, where applicable, the result. ("Standard system action" does not refer to any operating system but to standard action prescribed for the language.)

For the conditions OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, or FIXEDOVERFLOW, an interrupt action will always take place on occurrence of the condition unless the occurrence is in a calculation lying within the scope of a prefix specifying NO OVERFLOW, NO UNDERFLOW, NO ZERODIVIDE, NO CONVERSION, or NO FIXEDOVERFLOW. For the conditions SIZE, SUBSCRIPTRANGE, or CHECK (identifier list), an interrupt will not take place on occurrence of the condition unless the occurrence is in a calculation lying within the scope of a prefix specifying the condition. (See "Prefixes," in Chapter 1).

For any other condition, whose name may not be used in a prefix, an interrupt always will result from the occurrence of the condition.

CLASSIFICATION OF CONDITIONS

The ON-conditions are classified as follows: computational conditions, input/output conditions, program-checkout conditions, programmer-named conditions, and system-action conditions.

The computational conditions are associated with data handling, expression evaluation, and computation.

The input/output conditions are associated with data transmission.

The program-checkout conditions facilitate debugging of programs.

The programmer-named conditions permit the programmer to use conditions of his own naming. These conditions are raised only by a SIGNAL statement.

The system-action conditions provide facilities to the programmer to extend the standard system action taken after the occurrence of a condition or at the completion of a program.

COMPUTATIONAL CONDITIONS

CONVERSION: This condition occurs when there is an error in data type conversion.

Result: Undefined.

Standard system action: Comment and continue.

FIXEDOVERFLOW: This condition occurs during fixed-point arithmetic operations if the results of these operations exceed N, the maximum field width as defined by the implementation. See SIZE for a related condition that occurs on assignment.

Result: Truncation on the left to size N.

Standard system action: Comment and continue.

OVERFLOW: This condition occurs when the exponent of a floating-point number exceeds the permitted maximum, as defined by the implementation.

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: Maximum positive value.

Standard system action: Comment and terminate execution.

SIZE: The condition arises when a fixed-point value is assigned to an item of fixed-point data, with a loss of high-order bits or digits.

The SIZE condition should be distinguished from FIXEDOVERFLOW that occurs during arithmetic calculations. A value too large for the field to which it is assigned will raise a SIZE condition on assignment, regardless of whether there was overflow in the calculation of the value. FIXEDOVERFLOW depends upon the size of fixed-point numbers allowed in the implementation. SIZE depends upon the declared size of the item of data receiving a value.

Result: Truncation on the left to declared size.

Standard system action: Comment and terminate execution.

UNDERFLOW: This condition occurs when the exponent of a floating-point number is smaller than the permitted minimum, as defined by the implementation.

The condition does not occur when equal numbers are subtracted (often call significance error).

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: Smallest positive non-zero value.

Standard system action: Comment and continue execution.

ZERODIVIDE: This condition occurs on an attempt to divide by zero. The condition does not distinguish between fixed-point and floating-point division; either can cause it.

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: Undefined.

Standard system action: Comment and terminate execution.

INPUT/OUTPUT CONDITIONS

ACCESS (filename): This condition is raised whenever a programming error prevents successful access of a record from the designated file. The particular error may be determined by means of the ONCODE built-in function.

Standard system action: Comment and terminate execution.

EDIT (filename): This condition is caused by an illegal character in the input data from a specified file.

Standard system action: Comment and terminate execution.

ENDFILE (filename): This condition is caused by an attempt to read past a file delimiter from the specified file.

Standard system action: Comment and terminate execution.

ENDGROUP (filename): This condition is caused by an attempt to read past a group delimiter from the specified file.

Standard system action: Comment and terminate execution.

ENDRECORD (filename): This condition is caused by an illegal attempt to read past a record delimiter from the specified file.

Standard system action: Comment and terminate execution.

FIELDOVERFLOW (filename): This condition is caused by an output item that is too large for the output field width specified. If numeric, the leading zeros are ignored.

Standard system action: Comment and terminate execution.

IDENT (filename): This condition is raised if the OPEN or CLOSE IDENT option does not compare with the label on the designated file. This applies only to the IDENT option form that specifies both data list and format list.

Standard system action: Comment and terminate execution.

NAME (filename): This condition is caused by an unrecognizable identifier on data-directed input.

Standard system action: Comment and terminate execution.

SEARCH (filename): This condition is caused by the inability to find the requested keyed record from the specified file.

Standard system action: Comment and terminate execution.

TRANSMIT (filename): This condition is caused by a transmission error on the specified file.

Standard system action: Comment and retry, and if unsuccessful after the standard number of retries, comment and terminate execution.

UNDEFINEDFILE (filename): This condition is raised when the specified file is not available.

Standard system action: Comment and terminate execution.

PROGRAM CHECKOUT CONDITIONS

SUBSCRIPTRANGE: This condition occurs when a subscript is evaluated and found to lie outside its specified bounds.

The condition does not distinguish between values that are too large and values that are too small.

Result: Undefined.

Standard system action: Comment and terminate execution.

CHECK (identifier list): A statement prefix specifying this condition may only be applied to PROCEDURE or BEGIN statements.

In the identifier list, each identifier is one of the following:

- a statement label
- an unsubscripted variable name representing a scalar, array, structure, or label variable
- an entry label

Each item in the list is, in effect, enabled independently.

Statement Label: For a statement-label identifier the condition is raised prior to the execution of the statement to which the label is prefixed. If the label is prefixed to a non-executable statement, no condition will be raised.

Variables: For identifiers representing variables, the condition is raised whenever the value of the variable, or any generation of any part of the variable, may have been changed by any statement within the scope of the prefix.

The condition will be raised by the explicit reference to an identifier ID in the circumstances listed below, where ID is:

- an identifier in the list
- an identifier representing a structure or element contained by, or containing, an identifier in the list

The reference to ID may be subscripted or qualified.

The condition will be raised for ID if:

1. ID appears on the left hand side of an assignment statement. (This applies to assignment BY NAME even if the identifier mentioned does not appear in the final expansion of the statement.)
2. ID is set as a result of a pseudo-variable or pseudo-array appearing on the left hand side of an assignment.
3. ID appears as the controlled variable of a DO statement (or ID is set as a result of a pseudo-variable appearing as the controlled variable of a DO loop).
4. ID appears in a data list on a READ or GET statement.
5. ID has the SYMBOL attribute and a data-directed READ or GET statement is executed.
6. ID appears as the second argument of a DISPLAY statement.
7. ID appears as a STRING option on a WRITE statement.
8. If a procedure is invoked and any of the following applies:
   a. ID is external
   b. The procedure knows ID
   c. ID is passed as an argument (and no dummy is created)
9. ID appears on a RESTORE statement.

However, the condition is NOT raised under any of the following circumstances:

1. If the value of a variable defined upon ID or upon part of ID changes value in any of the ways described above.
2. If the value of a variable upon which ID is defined changes value.
3. If a parameter which represents ID changes value.

Condition 8 is raised immediately after control is returned from the invoked procedure. If control is not returned to the point of call, the condition is not raised . If the procedure is invoked asynchronously, the condition is raised when the attaching task regains control.

All other conditions are raised immediately after the statement that caused the condition has been executed. This applies even if the statement is executed asynchronously. If the execution of the statement is not completed due to a function reference not returning a control to its point of call, then the condition is not raised. In case 3, the condition is raised before each iteration of the group or statement following the DO statement. It is not raised if this group or statement is skipped over.

Entry Labels: For an entry label, the condition is raised prior to each invocation of the entry label. The condition is raised only if the entry label is invoked by the name given in the ON list.

PROGRAMMER-NAMED CONDITIONS

CONDITION (identifier): The "identifier" is, in a sense, a condition-name created by the programmer. The condition can be caused to occur only by the execution of a SIGNAL statement for the named condition (see "The SIGNAL Statement," in Chapter 6, for an example of this condition).

SYSTEM ACTION CONDITIONS

FINISH: This condition occurs after a program is terminated (by any means) and after standard system action has been taken.

Standard system action: Comment and return control to the system.

ERROR: This condition occurs after the standard system action has been taken for any condition that causes execution to be terminated before the anticipated end.

Standard system action: Comment and terminate execution.

APPENDIX 4: PERMISSIBLE KEYWORD ABBREVIATIONS

Abbreviations are provided for certain keywords. The abbreviations themselves are keywords and will be recognized as synonomous in every respect with the full keywords. The abbreviated keywords are shown to the right of the full keywords in the following list.

PROCEDURE	PROC
DECLARE	DCL
DECIMAL	DEC
BINARY	BIN
COMPLEX	CPLX
COMPLETE	CPLT
CHARACTER	CHAR
VARYING	VAR
POSITION	POS
INITIAL	INIT
INTERNAL	INT
EXTERNAL	EXT
AUTOMATIC	AUTO
CONTROLLED	CTL
DEFINED	DEF
ABNORMAL	ABNL
PRECISION	PREC
OVERFLOW	OFL
UNDERFLOW	UFL
FIXEDOVERFLOW	FOFL
SUBSCRIPTRANGE	SUBRG
ZERODIVIDE	ZDIV
CONVERSION	CONV

The characters that make up the 48-character set are same as those that make up 60-character set except for certain restrictions.

The following characters are not included:

Percent	%
Colon	:
Not	¬
Or	|
And	&
Greater Than	>
Less Than	<
Break character	_
Semicolon	;
Number sign	#
Commercial At sign	@
Question mark	?

The following three characters are replaced as indicated:

60-Character Set	48-Character Set
:	..
;	,.
%	//

The two periods which replace the colon must be immediately followed by a blank if the following character is a period.

The following operators, as used in the 60-character, set are replaced in the 48-character set by alphabetic operators as indicated:

60-Character Set	48-Character Set
>	GT
>=	GE
¬=	NE
<=	LE
<	LT
¬	NOT
|	OR
&	AND
||	CAT

In each case, one or more blanks must immediately precede the alphabetic operator if the preceding character would otherwise be alphameric, and one or more blanks must immediately follow if the following character would otherwise be alphameric. Thus, to indicate the comparison of the variables A6 and BQ2Y for inequality, one would write A6 NE BQ2Y, but not A6NEBQ2Y, A6 NEBQ2Y, or A6NE BQ2Y. As the equal symbol is usable, however, the comparison of these two variables for equality may be written A6=BQ2Y.

The word NOT is "reserved" in the 48-character set; that is, it must not be used as a programmer-specified identifier.

The break character, commercial at-sign, and number sign are not used and consequently may not be employed in identifiers.

(If more than one page number is given, the primary discussion is listed first.)